Sistemas de localización y medición de distancias basados en ultrasonidos: Estudio e implementación

Iván Tomás Rubio

Directora: Anna Calveras Augé

Abril 2010

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Agradecimientos

Este trabajo lo he creado empezando prácticamente de cero, sin conocimientos sobre ninguno de estos temas. Pese a ello, he recibido mucha comprensión y ayuda en el laboratorio, donde todos los que han podido me han echado una mano.

Por ello quiero agradecer a Anna Calveras por proponerme este trabajo y guiarme y aconsejarme a lo largo de la duración del proyecto, así como facilitarme la beca.

Gracias a Eliseo, Jordi Vilaseca, Jordi Casals, Javier Gálvez, Miguel, Toni, Jacobo, Guillermo y Ernesto por su compañerismo y amistad a lo largo de este tiempo. La estancia aquí ha sido mucho más agradable con vosotros.

Me gustaría hacer una mención especial a José Luis Ferrer, por su inestimable ayuda con el hardware, y a Marisa Catalán, por dirigirme, contestar todas las preguntas y ayudarme siempre que ha podido.

Gracias a mis amigos por aguantarme todos estos años y por todos los buenos ratos que me habéis hecho pasar. Yo no me habría aguantado a mí mismo.

Gracias a Verónica porque, incomprensiblemente, aún está conmigo, y siempre ha sido un gran punto de apoyo.

Finalmente, gracias a mi familia, especialmente a mis padres y mi hermana, porque sin ellos nada de esto hubiera sido posible.

Índice

Índice de ilustraciones

Índice de tablas

1 Introducción

Hay un incremento de interés en personalizar aplicaciones para el usuario, no solo a nivel global, sino a nivel local, lo cual pasa por ubicarlo con precisión. Esto permite todo tipo de aplicaciones avanzadas, como personalizar la publicidad según su situación, sistemas de seguridad avanzados o una planificación de mobiliario personalizada a las necesidades del público.

También, sobretodo en el campo de la automoción, se ha presentado mucho interés por sistemas capaces de situar obstáculos u otros vehículos cercanos. Esto podría permitir, entre otras cosas, reducir accidentes, aumentar la seguridad en la conducción o, enfocándolo hacia el espectáculo, ayudar a monitorizar carreras.

Así pues, este proyecto se enfoca con la intención de crear un sistema de localización de blancos y medición de distancias para dos situaciones concretas, representadas en dos proyectos asociados a UPC, Arenalife y Vodafone, cada uno con unas exigencias específicas.

En concreto, el primer proyecto, de Arenalife, requería de un método por el que localizar la distancia relativa entre ciclistas en una carrera, lo que necesita una solución de bajo consumo y portátil, de poco peso, lo cual limita la elección del hardware. El segundo proyecto, de Vodafone, estaba enfocado al conteo y la ubicación de personas en una sala de conferencias. Para esto se necesita de un sistema que sea capaz de discernir entre los elementos fijos de la habitación y los elementos de interés, en este caso, las personas.

En ambos casos el principal objetivo es crear la programación necesaria capaz de detectar la distancia entre un punto, que se dotará de un nodo, y los blancos que estén dentro de su alcance. Fuera de los objetivos de este proyecto se encuentra el realizar un sistema centralizado que recopile toda la información y la procese para ubicar las posiciones dentro de un sistema de referencia.

Los objetivos de este proyecto se tenían que adaptar a ambos ámbitos, y por extensión, a ambos tipos de blancos:

- Cuando los blancos puedan portar un dispositivo identificador. Este caso englobaría el primer proyecto, un pelotón de ciclistas, donde cada uno podría llevar instrumentación ligera en el casco o en la bici, para localizarse mutuamente.

  Al permitir la posibilidad de incluir un elemento identificador en cada blanco, es fácil asignarles un identificador específico a cada uno de ellos. En este caso la actuación sería principalmente al aire libre y con blancos en movimiento.

- Cuando los blancos no puedan portar un dispositivo identificador. Este caso sería el correspondiente al segundo proyecto, y serviría para controlar la situación de la gente que entra en una tienda o en una sala de reuniones, puesto que sería inviable dotarles de ningún tipo de dispositivo en este caso.

  Como no se les puede dotar de ningún tipo de instrumentación, lo mejor será actuar de manera lo menos intrusiva posible. Esto implicará, no obstante, menos precisión y la incapacidad de decidir quién es un blanco concreto. En este caso el foco sería entornos cerrados con blancos más o menos estáticos.

En primer lugar se trabajó para cumplir el primer objetivo, correspondiente al proyecto de Arenalife, pero posteriormente surgió la oportunidad de colaborar también en el proyecto de Vodafone y se puso como prioridad al tener unos requisitos más definidos y una fecha de entrega más cercana. Es decir, finalmente es de mayor interés la capacidad de localizar personas en un lugar cerrado, pero aún y así es muy importante mantener un nivel de compatibilidad adecuado para poder utilizarse para el primer objetivo, la localización de objetivos móviles en un entorno abierto.

En este proyecto, en primer lugar se estudiarán dos posibles tecnologías capaces de localizar en ambos casos arriba expuestos, en concreto, ultrasonidos y Ultra Wide Band. Lamentablemente, esta segunda tecnología no ha estado disponible en la duración del proyecto, por lo que se ha descartado su uso. También se verán las principales características del hardware utilizado, así como se analizarán diferentes algoritmos de detección de picos para localizar el instante de llegada de un pulso de ultrasonidos.

En segundo lugar se pasará a la preparación del entorno de trabajo para poder programar el hardware disponible. Se programará un ejemplo para ver que tanto el conversor analógico-digital de los nodos como la comunicación con el PC funciona correctamente.

Finalmente, se programará una solución que permita la localización de blancos bajo los dos supuestos anteriormente mencionados. Primero se programará para que emisor y receptor se encuentren en nodos separados y sean capaces de comunicarse vía radio y calcular la distancia entre ellos mediante un pulso de ultrasonidos. Después se modificará para el caso en que el emisor y receptor se encuentran en el mismo nodo y deben encontrar el blanco comportándose como un sonar, es decir, midiendo la distancia por el tiempo que tarda un rebote de la señal en impactar con el blanco.

2 Estado del arte de la tecnología para aplicaciones de localización

Como primera aproximación al proyecto lo interesante era comprobar el estado actual y pasado de las diferentes tecnologías y proyectos que podrían ser utilizados para la localización de tal manera que se ajusten a los parámetros comentados en la introducción.

En primer lugar, se explicarán las principales características detrás de las dos tecnologías más importantes en este sector, ultrasonidos y Ultra Wide Band (UWB). En segundo lugar, se pasará a hacer una pequeña descripción de los proyectos más relevantes que utilizan estas tecnologías. Después de comentar las conclusiones sobre estos proyectos, como punto final, se hablará de la necesidad de implementar un algoritmo de detección de picos en la programación y las diferentes alternativas que existen.

2.1 La elección entre Ultrasonidos y Ultra Wide Band

2.1.1 Ultrasonidos

Se conocen como ultrasonidos las ondas acústicas superiores a la máxima frecuencia auditiva perceptible por el ser humano. Nominalmente esto incluye cualquier frecuencia sobre 20 kHz, pero habitualmente se trabaja entre 1 y 20 Mhz.

Las principales aplicaciones de los ultrasonidos frente a las ondas de radio convencionales en el ámbito de la localización vienen del hecho que la velocidad de los ultrasonidos es aproximadamente la velocidad del sonido ($v_s \approx 343\, m/s$), mientras que las ondas de radio viajan a la velocidades de la luz ($c = 10^8\, m/s$). Esto permite mandar un mensaje radio y un pulso de ultrasonidos en el mismo instante desde un emisor y ver la diferencia entre el tiempo de llegada del paquete radio y del pulso de ultrasonidos en el receptor, con lo que se puede extraer la información sobre la distancia relativa. Esto se conoce como Time Difference of Arrival (TDoA). En este caso, la distancia se puede medir como:

$$d = v_s(t_s + t_c) = v_s(t_s + \frac{d}{c}) = v_s t_s + d\frac{v_s}{c}$$

$$d - d\frac{v_s}{c} = d(1 - \frac{v_s}{c}) = v_s t_s \Rightarrow d = \frac{v_s t_s}{1 - v_s/c}$$

$$\frac{v_s}{c} \ll 1 \Rightarrow d = v_s t_s$$

Donde v_s es la velocidad del sonido, t_c es el instante en que llega el paquete radio al blanco, t_s es el tiempo que tarda en recibir el pulso de ultrasonidos el blanco una vez ha recibido el aviso con el paquete radio y c es la velocidad de la luz. Como la velocidad de la luz es muy superior a la del sonido se puede decir que, a efectos prácticos, el paquete radio no afecta en la medición y por tanto sólo depende del pulso de ultrasonidos.

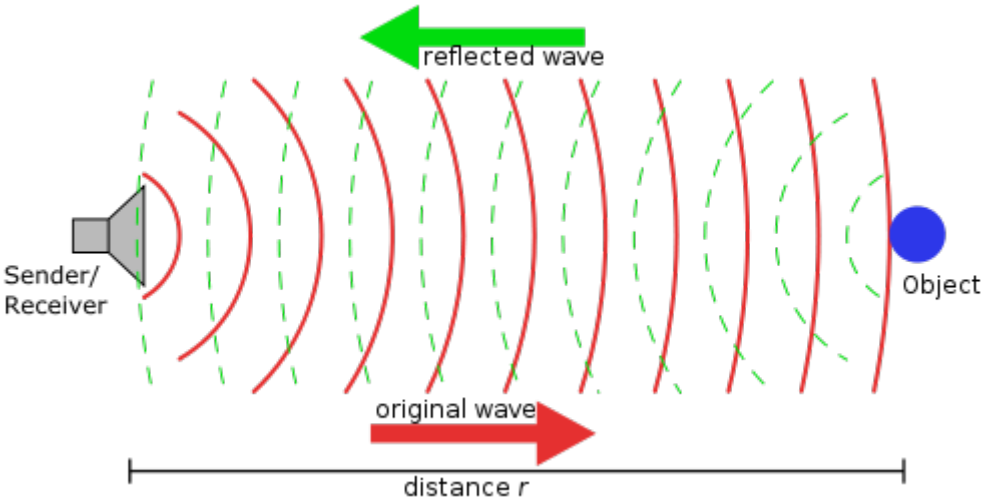

Fig. 1: Funcionamiento de un sonar
(fuente: http://en.wikipedia.org/wiki/File:Sonar_Principle_EN.svg)

También es posible utilizar simplemente el tiempo de llegada (Time of Arrival, ToA) del pulso de ultrasonidos para calcular distancias, en caso de que el emisor y el receptor sepan a la vez cuando se ha emitido el pulso de ultrasonidos, como por ejemplo en el caso de un sonar, ilustrado en la figura 1. En este caso, para calcular la distancia es muy sencillo, sólo se ha de tener en cuenta que la señal rebota en el blanco y vuelve, con lo que recorre dos veces esta separación:

$$d = \frac{v_s t}{2}$$

Donde v_s es la velocidad del sonido y t es el tiempo que tarda la onda en llegar hasta el blanco, rebotar y volver de nuevo emisor.

Para la mayoría de aplicaciones de localización con ultrasonidos el emisor y el receptor están separados, y deberán tener línea de visión directa entre ellos (Line of Sight, LOS). Esto implica que ambos deben estar encarados, de manera que la señal llega de uno a otro directamente, sin rebotes. No obstante, es posible utilizarlo en situaciones sin línea de visión directa (Non-Line of Sight, NLOS), aunque se ha de lidiar con alcances mucho menores debido a la disipación de energía de las ondas acústicas al rebotar en la mayoría de superficies.

Al usar ultrasonidos se ha de tener en cuenta que muchas acciones cotidianas pueden generar un nivel importante de ruido en la frecuencia de ultrasonidos, pese a no ser perceptibles para los humanos, como por ejemplo abrir una puerta o las rozaduras entre metales.

2.1.2 Ultra Wide Band

Se dice que se está utilizando Ultra Wide Band cuando se transmite con un ancho de banda tal que el ancho de banda relativo B_f es mayor que el 25%.

$$B_f = \frac{B}{f_c}$$

Donde f_c es la frecuencia central y B el ancho de banda total utilizado. Con esto se consigue que la distribución de energía se reparta por un rango de frecuencias muy grande, con lo que la densidad espectral es muy pequeña. Esto provoca que sea prácticamente imposible que se produzcan interferencias perceptibles entre UWB y otras señales de radio que trabajen en el mismo lugar, ya

que la potencia de las otras señales estará concentrada en una banda y verá al UWB como ruido blanco. Además, visto en clave temporal, al tratarse de pulsos con unas duraciones extremadamente pequeñas, típicamente medidas en picosegundos, es prácticamente imposible que coincidan con otros pulsos, incluyendo UWB. Al ser casi imposible que se solapen los pulsos, se puede decir que son prácticamente inmunes a las interferencias.

Se puede aplicar esta tecnología a la localización de manera similar a un Radar, es decir, enviando un pulso de UWB y midiendo el tiempo hasta que retorna, o bien sincronizando emisor y receptor y midiendo únicamente el tiempo que tarda el pulso en ir del primero al segundo. Debido a su resistencia contra las interferencias, es más fácil utilizarlo en condiciones NLOS que en el caso de los ultrasonidos.

Midiendo la potencia de la señal recibida (Received Signal Strength, RSS) y mediante un *array* de antenas, es posible medir el ángulo de llegada (Angle of Arrival, AoA) del pulso recibido. En aplicaciones de localización, esto puede ayudar considerablemente a situar la posición del emisor.

Dado que tienen un ancho de banda muy elevado, no es necesaria una gran potencia de señal para obtener buenas relaciones señal a ruido y, por consiguiente, conseguir velocidades altas de transmisión. Por otro lado, al ocupar un rango de frecuencias tan elevado, transmitir con mucha potencia podría causar problemas en otros sistemas que funcionen dentro de este rango, ya que verían un incremento del ruido.

Pese a las ventajas, se ha de tener en cuenta que el coste general de los equipos de UWB será más elevado, debido tanto al tipo de emisores y receptores necesarios como a la velocidad de procesado necesaria para trabajar con pulsos tan estrechos: se requieren tolerancias de error muy bajas a la hora de tener en cuenta el sincronismo entre los diferentes componentes.

2.2 Proyectos más relevantes basados en ultrasonidos

2.2.1 Cricket

El MIT actualmente fabrica una serie de dispositivos denominados Cricket[1]. Se usan principalmente en situaciones *indoor*, es decir, dentro de edificios, donde se colocan unos dispositivos denominados *beacons* ("faros") que envían señal de radiofrecuencia (RF) y ultrasonidos (US). Los objetos a localizar se proveen de *listeners* ("escuchadores") que reciben las señales emitidas por los *beacons* y las procesan para determinar en qué posición con respecto a la red de *beacons* se encuentran. Inicialmente solamente era posible saber a qué *beacon* estaban acoplados, de manera que la resolución era zonal, pero posteriormente se realizó una nueva versión que permitía, mediante trilateración, el conocer la posición con precisión de hasta 1 a 3 cm[2], más útil para el objetivo de este proyecto.

En la figura 2 se puede ver un nodo Cricket, con el puerto serie para conectarlo al PC a la derecha, el emisor y receptor de ultrasonidos a la izquierda y un conector de 51 pins en la parte de abajo para conectar placas de sensores.

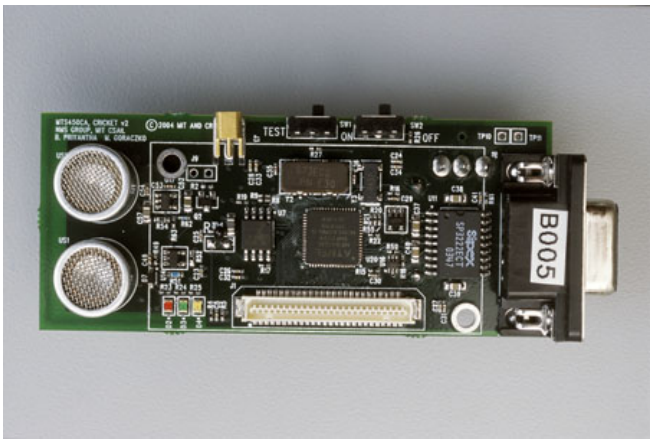

Fig. 2: Nodo Cricket
(fuente:
http://cricket.csail.mit.edu/pictures/cricketv2.jpg)

El principio de funcionamiento en el que se basa es en el TDoA entre la onda de RF y la de US. Así es posible calcular a qué distancia está un determinado *listener* de los *beacons* más cercanos, datos con los cuales es posible extraer su posición absoluta en el sistema. Para evitar colisiones, en un momento dado solamente emite un *beacon*, de manera tal que el pulso RF contiene un identificador único para dicho emisor y el pulso de US sin información salen a la vez, para posteriormente poder comparar tiempos. El *listener* sabe de donde proviene el pulso de US debido a que éste llegará mientras aún se emite el pulso de RF. Para evitar los errores persistentes a la hora de transmitir el pulso de RF y no implementar un sistema completo de acceso al medio, en lugar de utilizar una cronología determinista para la transmisión, se elige el tiempo de transmisión siguiendo una distribución uniforme dentro de un intervalo. El intervalo se elige de tal manera que se cumple el compromiso esperado entre tiempo medio para encontrar la posición y posibilidad de colisión.

No obstante, debido a que en un momento dado el *listener* solamente escucha a un receptor, a la hora de localizar objetivos móviles no podrá conocer su posición hasta tener al menos dos lecturas más de dos receptores distintos al inicial. Adicionalmente, si alguna de estas lecturas no es válida, el tiempo para resolver la posición aumenta. La manera que tiene Cricket de aliviar este problema es el uso del filtrado Kalman extendido, que mantiene una estimación de la posición y la velocidad

actuales y las va ajustando con los datos reales que va recogiendo, basándose en la suposición que la velocidad entre pulsos se mantiene constante. Esto no siempre tiene porqué ser cierto, lo que provoca un error en la estimación y, por tanto, en la matriz de covarianza. Cuando este error es demasiado elevado, la matriz se resetea y se empieza la estimación desde cero con los últimos valores recogidos, pero para realizar estas medidas, es necesario volver al *listener* capaz de emitir activamente. Se genera un pulso en un *time slot* donde todos los *beacons* escuchan, procesan y envían información de vuelta con una estimación de la distancia. Como esta información es tomada a la vez en todos los receptores, el ajuste de la posición ya no conlleva el problema de sólo haber escuchado un valor.

2.2.2 Active Bat

Un proyecto que también basa su modo de funcionamiento en ultrasonidos es el denominado Active Bat[4][5]. Sucesor de Active Badge, que localizaba por infrarrojos con una resolución de una habitación, el Active Bat de AT&T ofrece una resolución de hasta 3 cm en las tres dimensiones. Este sistema utiliza una matriz de sensores ubicados en el techo de la habitación separados 1.2 m entre ellos (en rojo en la figura 3), de manera que cuando se desea localizar a un nodo (en azul en la figura 3), denominado Bat, se le envía una señal por radiofrecuencia con su identificador y éste pasa a transmitir una señal de US que es recibida por los sensores y se procesa la posición con trilateración de manera similar a Cricket. Esta señal RF también puede transmitir un código para que el Bat se ponga en modo de espera durante un tiempo determinado, para ahorrar baterías.

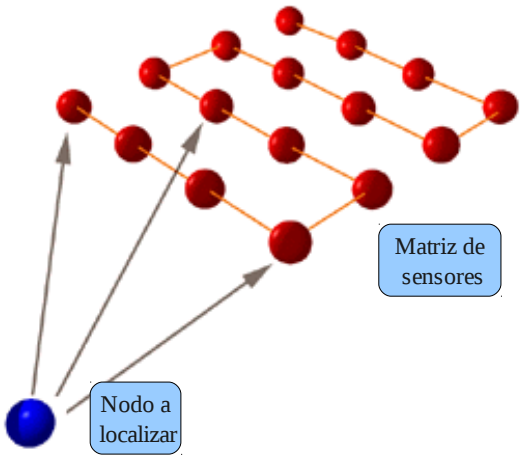

Fig. 3: Triangulación con Active Bat (fuente: [3])

Al pedir la emisión del pulso a un único nodo cada vez es fácil aumentar el número de nodos en el sistema ya que no habrá que tener en cuenta colisiones de ningún tipo. El hecho que todos los receptores del techo escuchen a la vez mientras el Bat transmite hará que varios sensores tengan su posición relativa al nodo, y mientras este número de sensores sea 3 o más, la posición se conocerá sin incertidumbre. En cambio, esto también provocará que contra más nodos haya para localizar, más tiempo se tardará en localizarlos a todos al tener que ir de uno en uno. Además, el tener que disponer de una matriz de sensores en con una distribución determinada y conocida para detectar la posición de los nodos limita severamente la utilidad de este sistema.

2.2.3 Constellation

Constellation[6] es un proyecto de InterSense enfocado al seguimiento de movimiento, trabajando

de manera similar a un sistema de navegación inercial (INS). Con un sensor inercial y tres módulos de medición de distancia por ultrasonidos se consigue conocer en cada momento posición y velocidad de un móvil.

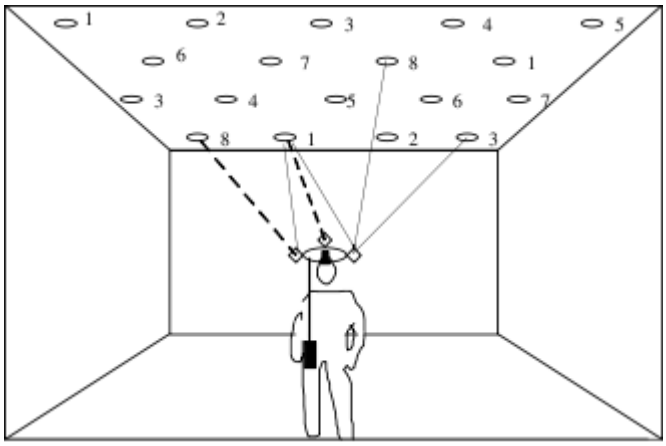

Fig. 4: Idea general de Constellation
(fuente: [6])

Los módulos de medición de distancias se comunican con una constelación de *beacons* en posiciones conocidas y piden mediante infrarrojos la emisión de un pulso de ultrasonidos a un emisor concreto, con tal de detectar la distancia hasta ese emisor mediante el cálculo de TOF (Time of Flight). Los emisores actuarán de uno en uno y según un código único embebido dentro del pulso de infrarrojos que los identificará. Las medidas de distancias se pasarán por un filtro de Kalman extendido que ajustará los valores de posición y orientación con ayuda de las predicciones realizadas con el sensor inercial.

En la figura 4 se muestra como es el esquema de funcionamiento esperado. El aparato de la cabeza con 3 nodos se comunicaría con la matriz del techo para saber localización y orientación, así que se tienen 6 variables de posición a determinar. Para hallarlas completamente, son necesarias 6 medidas de rango, conectando entre al menos 3 receptores y 3 emisores. No obstante, dos grados de libertad pueden resolverse estabilizando con respecto a la gravedad, así que teóricamente con solo 4 LOS emisor-receptor se podría resolver completamente el sistema.

Como ventajas, al estar basado principalmente en el trabajo del sensor de inercia, se tiene alta predecibilidad y suavidad en la transición entre posiciones detectadas. Además, si se tienen suficientes emisores ocupando una zona adecuadamente extensa, tiene una precisión teórica de unos 4 mm.

Como inconvenientes, se necesitan una serie de *beacons* con posición conocida para poder realizar los cálculos de manera válida, así como dotar al objeto que se desea posicionar con varios sensores. Además, el uso de infrarrojos puede no resultar adecuado, aunque esto pueda cambiarse por un sistema de RF.

2.2.4 Dolphin

El proyecto Dolphin[7] (Distributed Object Locating System for Physical-space Internetworking) es un proyecto de la Universidad de Tokyo y tiene como objetivo eliminar la pre-configuración manual de las estaciones de referencia que son necesarias para ubicar los móviles en sistemas como Active Bat o Cricket. Para ello, propone un algoritmo tal que la localización global en el sistema se obtiene mediante el cálculo distribuido de las posiciones.

La determinación de distancias relativas entre nodos es prácticamente igual que el de Cricket, es

decir, se envía a la vez una señal de RF y una de US y se mide el TDoA entre ambas. La novedad del algoritmo es que simplemente teniendo un pequeño número de nodos conocidos con posiciones fijas y calculadas se puede saber donde están los demás nodos, sin ser necesario recibir directamente la señal de estos nodos de referencia. El funcionamiento de este algoritmo *hop-by-hop*, mostrado en la figura 5, se basa en que cada nodo tiene una ID única, una tabla de posiciones y una lista de nodos. A un nodo inicial de referencia se le da el rol de maestro, y pide a otro nodo que transmita desde su posición para calcular la distancia. Debido a esta transmisión, todos los nodos receptores que están en LOS con este nodo transmisor actualizan ahora su tabla de posiciones con la posición respecto a este nodo. Si esta transmisión ha servido para saber con exactitud la posición del nodo receptor en el sistema de referencia, se envía un mensaje por RF indicando su posición y su identidad para que los otros nodos actualicen las tablas. Los nodos que tienen su posición determinada en el sistema global son elegibles como nodos maestros, de manera que, lentamente, los nodos acaban sabiendo todos su posición sin tener que estar directamente en contacto con los de referencia.

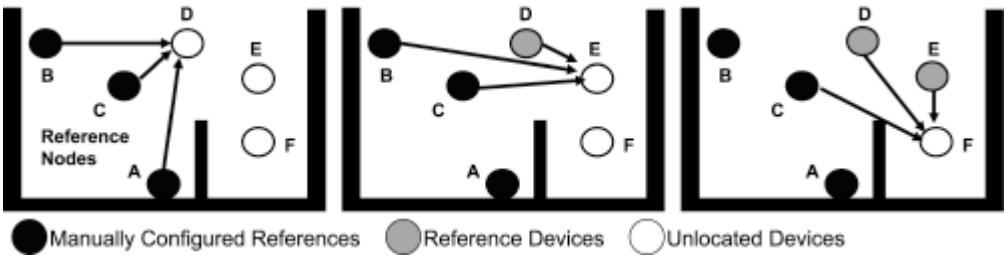

Fig. 5: Funcionamiento del algoritmo iterativo de Dolphin (fuente: [7])

Este sistema distribuido tiene la ventaja que es fácilmente escalable, puesto que no es necesario tener una infraestructura relativamente grande ni seguir un determinado patrón de despliegue. Como desventajas, se encuentra que la posición podrá tener errores elevados debido a que no se recibe la posición de manera actualizada, sino solo en los casos en que recibe señal de un nodo transmisor.

2.2.5 AHLoS

AHLoS[8] (Ad-Hoc Localization System) es un sistema de localización que, como Dolphin, calcula la posición de los nodos de manera distribuida e iterativa. Para ello, en el sistema se tendrán una serie de nodos conocidos que actuarán como *beacons*, y unos nodos a localizar en ubicación desconocida. Los *beacons* emiten su posición y un pulso de US, que son capturados por los nodos cercanos, también mediante la técnica de TDoA. Estimando la distancia a tres o más nodos conocidos, un nodo a localizar se convierte en un *beacon* que a su vez podrá ayudará a localizar el resto de nodos que falten. Así, paso a paso, se localizan todos los nodos que reciban señales suficientes.

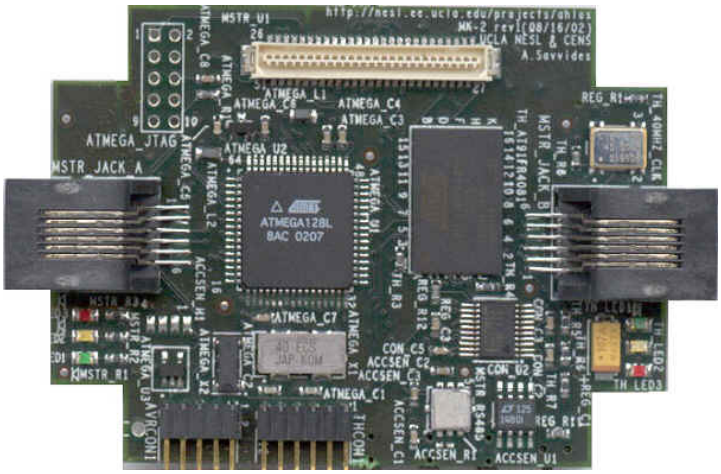

Fig. 6: Nodo Medusa Mk.2
(fuente: [9])

Este proyecto también cuenta con nodos de fabricación propia, denominados Medusa, mostrados en la figura 6, que constan de una radio con un alcance de 20 m y varios puertos para conectarles sensores adicionales así como permitir la comunicación con un PC.

Al tratarse, como en el caso de Dolphin, de un sistema en el que la posición se procesa de manera distribuida, las ventajas y los inconvenientes serán prácticamente los mismos; como ventajas, una gran escalabilidad y sencillez de infraestructura, y como desventajas, los errores de estimación que se producirán al moverse los nodos.

2.2.6 Buzz

El sistema Buzz asíncrono[10], sucesor de [11], no requiere de una conexión central para emitir señales de control, si no que utiliza el efecto Doppler para evitar tener que utilizar un canal adicional de radiofrecuencia. Al no tener esta información adicional, la manera de distinguir entre las diferentes fuentes es asignando a cada emisor un período único entre pulsos emitidos. Este período se elige de manera que, para la velocidad máxima de movimiento esperada entre sensores en el sistema, sea muy difícil que se confunda el origen.

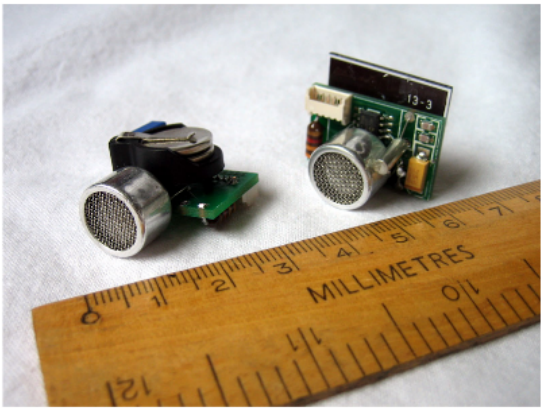

Fig. 7: Nodos Buzz asíncronos
(fuente: [10])

El algoritmo empareja cada pulso nuevo con la secuencia que tiene en memoria, y analiza las combinaciones posibles. Si se trata de una señal válida, almacena en memoria posición y velocidad, utilizando un filtro de Kalman con hipótesis múltiples para registrar la velocidad y posición. Con

cada nuevo pulso de entrada, se comprueban las hipótesis actuales para confirmarlas o descartarlas según sea necesario. Con estos valores, es posible predecir cuando debería llegar el siguiente pulso, y con la diferencia entre el tiempo esperado de llegada y el tiempo real, se pueden modificar los valores de memoria para reflejar el cambio.

Para descartar o validar las hipótesis, se utiliza un criterio de verosimilitud basado en la distancia de Mahalanobis. Si la estadística χ^2 cae dentro de unos parámetros, se considera como posible solución de las hipótesis. Contra más pulsos se hayan recogido, más fácil será descartar o validar los diferentes *beacons* como emisores reales de las señales.

Este proyecto ofrece una precisión de unos 20 cm y es capaz de distinguir movimiento además de posición, pero no es adecuado para los objetivos del proyecto debido a que la frecuencia de actualización es demasiado baja y la sensibilidad es demasiado elevada. Además, la falta de una manera de distinguir NLOS hace que no sea adecuado puesto que será necesario conocer los rebotes y de donde vienen.

2.2.7 Ultrasonidos de Banda Ancha

Hay también soluciones, como la propuesta en [12], que se basan en ultrasonidos de banda ancha. La idea detrás de utilizar este sistema en vez de la tradicional banda estrecha es obtener ventajas adicionales, como el hecho que varios emisores puedan transmitir a la vez, un aumento de resistencia al ruido y posibilidad de codificación de identificación en el sistema.

Para conseguir esto se utiliza una codificación DS-CDMA (Direct Sequence – Code Division Multiple Access) de manera que cada emisor se corresponde con una secuencia pseudoaleatoria perteneciente a los denominados “códigos áureos”, que tienen alta autocorrelación y baja correlación cruzada para, teóricamente, posibilitar el hecho que varios emisores trabajen a la vez y aún y así poder distinguirlos individualmente en el receptor.

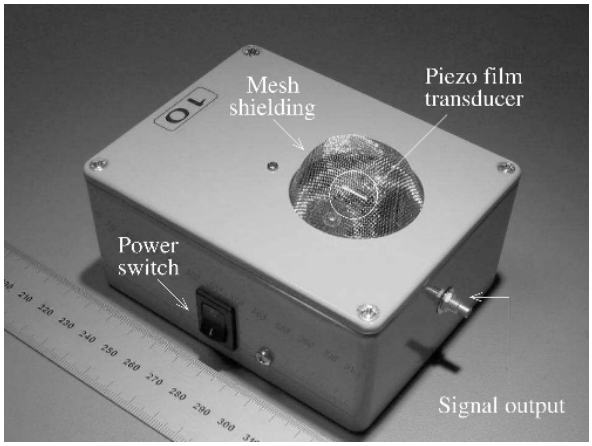

Fig. 8: Prototipo de Hazas y Hopper con transductor piezoeléctrico (fuente: [12])

No obstante, para poder dar estas prestaciones se necesita de unos emisores y receptores capaces de operar en banda ancha, lo cual no es el caso si se buscan dispositivos comerciales de coste reducido. Para ello, se utilizan transductores de piezopolímeros, como los utilizados en las aplicaciones médicas de ultrasonidos. Como contrapartida, estos transductores tienen una baja eficiencia.

Para descodificar la secuencia codificada se ha de tener en cuenta que aunque idealmente no debería haber correlación cruzada entre diferentes emisores, en el caso real si existe este factor, y por tanto se puede dar el denominado efecto *near-far* por el cual se degrada la señal de entrada por la

diferencia de potencia entre un emisor cercano y uno lejano al receptor. La manera elegida para lidiar con este problema es el uso de cancelación de interferencias sucesivas (SIC, Successive Interference Cancellation), también conocido como el algoritmo de Viterbi.

Este proyecto no es interesante para el caso a estudiar debido principalmente a que los aparatos utilizados son de dimensiones demasiado grandes (60 mm x 94 mm x 15 mm), como puede verse en la figura 7, y se busca una solución más general.

2.3 Proyectos más relevantes basados en Ultra Wide Band

2.3.1 ScenSor (Decawave)

El proyecto ScenSor[13] de Decawave, basado en el estándar IEEE802.15.4, utiliza Ultra Wide Band (UWB) y está pensado desde cero para ser utilizado como sistema de posicionamiento en tiempo real (Real Time Location System, RTLS).

El principio de funcionamiento de este sistema es la medida de la diferencia del tiempo entre llegadas de los diferentes transmisores hasta el receptor que ha de localizar. Es posible hacerlo de esta manera debido a que al tratarse de banda extremadamente ancha las pendientes de subida y bajada de los pulsos pueden ser muy elevadas, de manera que hay más precisión al detectar el cambio en la transmisión.

Este sistema tiene una precisión de hasta 10 cm, menor que el Cricket, pero en cambio la recepción permite una actualización más continuada de la posición, y el rango de cobertura es superior. Además, como parte del estándar se contempla la recepción tanto LOS como NLOS. Esto aumenta la versatilidad en entornos cerrados.

2.3.2 Ubisense

La empresa Ubisense posee un proyecto de UWB[14] preparado específicamente para situaciones *indoor*. La configuración que se utiliza es dotar a los móviles que se desea detectar con unos *tags* activos, mostrados en la figura 9, que transmitirán unos pulsos cuando algún sensor de la red de sensores, ubicada en posiciones conocidas dentro del recinto a monitorizar, se lo pida. Los sensores se agruparán en células, en una arquitectura parecida a la comunicación móvil por satélite, y dentro de cada célula habrá un sensor maestro que coordinará la actuación entre todos los sensores de su célula y mandará por Ethernet o WiFi las posiciones de los *tags* que se encuentren dentro de su sección.

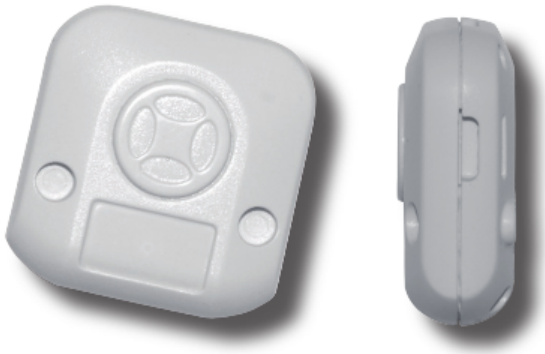

Fig. 9: Compact Tag de Ubisense (fuente: http://www.ubisense.net/pdf/fact-sheets/products/hardware/UbisenseSeries7000Compacttag090624EN.pdf)

Con la información de los sensores se puede calcular la posición del con una precisión de hasta 15 cm, actualizando esta posición hasta 20 veces por segundo. Adicionalmente, los sensores poseen un array de antenas que permite conocer el ángulo de incidencia con el que llega la señal. Esto se puede usar para obtener mejores medidas sin necesitar un número elevado de sensores detectando el pulso del *tag*. Los *tags* poseen un detector de movimiento que hace que se emita el pulso automáticamente si se detecta una aceleración puntual, para avisar al sistema que se está iniciando un movimiento. Cuando el sistema global conoce las posiciones y velocidades, ajusta la tasa de

actualización automáticamente para no pedir más pulsos de los necesarios con tal de conservar las baterías de los *tags* el máximo de tiempo posible.

2.3.3 Posicionamiento utilizando Round-Trip Transmission

La Universidad Soongsil de Seúl Según propone un algoritmo[15] que contempla la detección de un *tag* utilizando 3 *beacons*, pero usando algo distinto al TDoA, pese a asegurar que las prestaciones son similares.

En este caso lo que se tiene es 3 *beacons* con posiciones conocidas, todos conectados a una unidad de proceso, como puede verse en la figura 10. Uno de estos será el *beacon* maestro y los otros dos actuarán como esclavos. El maestro lanza un pulso, de manera que se guardan en los otros *beacons* el tiempo pasado desde que se envía, conocido por la información de sincronización proveída por la unidad de proceso. El *tag* entonces, pasado un tiempo T, enviará un pulso que será recibido por los tres *beacons*, que darán la información a la central de proceso para que calcule, según estos tiempos, la posición del *tag*.

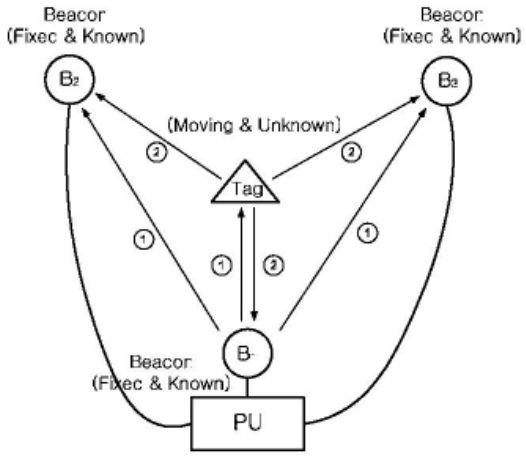

Fig. 10: Esquema del funcionamiento del algoritmo de localización por RTT (fuente: [15])

Este algoritmo teóricamente se comporta de manera similar al clásico ToA solo que no requiere de la complejidad adicional que supone tener una sincronización estricta y más dificultad de cálculo. Además, el consumo de batería es hasta tres veces menor debido al uso bajo demanda que se realiza.

2.3.4 Fusión entre TDoA y AoA

En [16] se propone un método para la localización de jugadores de fútbol en un estadio mediante UWB. Para ello, se compara el clásico TDoA con la mezcla de TDoA y AoA. Esta última opción utiliza métodos de fusión de datos para conseguir más precisión y calidad de servicio en el sistema sin aumentar mucho la complejidad de la implementación, comparándolo con otras técnicas clásicas.

En concreto, se le da a cada jugador un *tag* RFID que emitirá una señal que, mediante pseudo-ruido, se expandirá para que su espectro sea más elevado. Esta secuencia de pseudo-ruido será única para cada jugador para distinguirlos y combatir el efecto Near-Far que se produce cuando un transmisor no deseado está mucho más cerca del receptor que el transmisor deseado. El resultado de esto pasa por un modulador de frecuencia de onda continua (Frequency Modulated Continuous Wave), que

ayudará a posicionar al jugador, ya que al tratarse de UWB, se tiene la ventaja de disponer de un gran rango de frecuencias. Una vez hecho esto se pasará al procesado de datos mixtos TDoA/AoA y luego se calculará la posición exacta mediante hipérboles y un estimador “two-step Least-Square (LS) Estimator”, es decir, un estimador de mínimo cuadrático de dos pasos, para tener una carga de computación aceptable y una solución suficientemente precisa.

Se realizan una serie de simulaciones con tres antenas móviles como receptoras, y según los resultados se comporta mejor el método de fusión que únicamente utilizando TDoA incluso cuando no hay LOS con una o incluso con dos de estas antenas móviles. Como contrapartida, el tiempo de cálculo de la posición es ligeramente más elevado y requiere de una mayor complejidad al necesitar antenas receptoras móviles.

2.4 Conclusiones

Una vez analizadas las propuestas anteriores, queda compararlas con lo que realmente se busca para ver si alguna se ajusta al proyecto.

Como se ha comentado en la introducción, se busca que funcione principalmente en escenarios cerrados, como por ejemplo una sala de conferencias. No obstante, debe ser posible la adaptación a entornos abiertos donde varios móviles se localizarían entre ellos, como por ejemplo en el caso de unos ciclistas.

Para el primer caso sería posible tener una serie de sensores en unas posiciones determinadas de antemano, como requieren la mayoría de proyectos mencionados en el apartado anterior, pero por la misma aplicación que se busca (monitorizar personas, ocupación, etc), sería imposible o poco práctico dotar a cada blanco de un *tag* o similar. Para el segundo caso, en entornos móviles, es posible e incluso quizá necesario dotar de algún tipo de dispositivo a cada blanco, pero entonces sería prácticamente imposible tener una serie de posiciones fijas con las que comparar, por la misma naturaleza de la aplicación a desarrollar.

Por ello finalmente se decide realizar una solución propia lo más ajustada posible a los requisitos. Debido a que se dispone en esta misma escuela de tecnología de ultrasonidos probada y empleada en otros proyectos, no se dispone de material UWB y debido a que el coste es más reducido, tanto de transductores como procesadores necesarios, se decide por el uso de ultrasonidos en lugar de UWB. No obstante, en caso de que la escuela hubiera estado dotada de antemano de los elementos Ultra Wide Band necesarios o el precio hubiera sido más contenido, el proyecto se habría intentado adaptar para comparar las prestaciones de uno y otro. Concretamente, se espera que si se cambia la tecnología de ultrasonidos a UWB posteriormente, la precisión y el alcance de localización aumenten.

Como es difícil crear una solución global, se intentará enfocar prioritariamente hacia el escenario principal, monitorizar una sala. Esto implica, entre otras cosas, suprimir los sensores en el blanco y basar la localización enteramente en el rebote de la señal en éstos.

Así, será necesario procesar la señal proveniente del transductor de ultrasonidos para detectar los rebotes, que vendrán dados en forma de picos. Debido a esto, en el siguiente apartado se pasarán a analizar varias propuestas de detección de picos aplicables al problema y a elegir la más adecuada.

2.5 Detección de picos

Como se ha comentado, el funcionamiento final se espera similar al de un sonar, es decir, lo que se hace es emitir un pulso de ultrasonidos y se pasa a escuchar los rebotes durante un tiempo a determinar, de manera que se localizan los blancos por el tiempo que tarda en escucharse la señal de vuelta. La lectura de esto es un pulso de aproximadamente la misma duración que el enviado, retrasado un tiempo igual al doble del tiempo que tardaría la velocidad del sonido en recorrer la distancia que separa al emisor del blanco. Simplemente sabiendo en qué momento se recibe un rebote se tiene suficiente información para conocer la distancia del nodo a los blancos.

Para saber el instante en que se ha producido el rebote, la mejor manera es detectar cuando ha habido un pico en la señal para realizar el cálculo de distancia. A continuación se pasa a analizar varias propuestas encontradas en la bibliografía de algoritmos de detección de picos para hallar el más adecuado.

2.5.1 Propuestas de detección de picos

2.5.1.1 Umbral de detección

Con este sencillo método la manera de detectar los picos es asignando un umbral, de manera que cada vez que se cruza este umbral en sentido descendente se dice que se ha detectado un pico.

Este método tiene la ventaja de una extrema sencillez de cálculo, puesto que se requiere solamente de una comparación, pero como desventaja principal tiene que si se solapan parcialmente dos o más pulsos y no se llega a cruzar el umbral no se detectará más que un pulso. Además, el ruido puede hacer que se detecte un pulso varias veces si se cruza el umbral en varias ocasiones en el descenso de la señal.

2.5.1.2 Detección de máximos por derivada

Otro método en la bibliografía, explicado brevemente en [17], es la detección de máximos mediante la derivada. Cuando la derivada de una función en un punto es cero indica que se ha detectado una singularidad en este punto, que puede ser un máximo, un mínimo o un punto de inflexión. Como se trata de una señal digital, una manera de aproximar la derivada es mediante la resta de una muestra y la inmediatamente anterior. En este caso, obtener cero es prácticamente imposible, pero si que, igual que en el caso de la derivada continua, si se obtiene un valor positivo indica que la señal está aumentando su valor, y un valor negativo, que está descendiendo. Por ello, un máximo no es más que cuando el valor pasa de aumentar a descender, y por tanto cuando se pasa de tener un valor de derivada positivo a uno negativo. En la figura 11 se puede ver como funciona la implementación digital de este mecanismo, que detecta los pasos por cero.

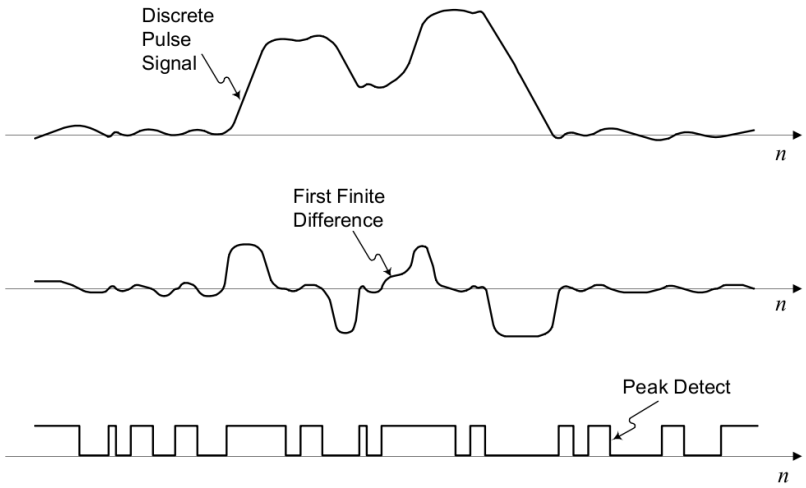

Fig. 11: Funcionamiento del detector por derivada (fuente: [17])

En este caso sí que se detectarían todos los picos, incluso habiendo solapamiento parcial entre ellos, pero como contrapartida es un método extremadamente sensible al ruido, ya que marcaría como picos cualquier instancia en la que la señal cambiara de aumentar a disminuir, lo que conllevaría, excepto en condiciones ideales, la detección de muchos más picos falsos que verdaderos.

2.5.1.3 Detector digital con umbral de ruido

En la propuesta de Jordanov et al[17] se da el montaje de un circuito digital para la detección de pulsos incluso cuando éstos se solapan parcialmente y en presencia de ruido.

En primer lugar, se necesita configurar el umbral de ruido. Esto determinará la facilidad con la que se decidirá si un pico se detecta como tal o si se descarta. El algoritmo entonces se basa en ir alternando dos modos de funcionamiento: buscar máximo y buscar mínimo. Cuando se busca máximo, se examina si el siguiente valor es mayor que el actual o si es menor pero con un margen suficiente para que el cambio no lo haya producido el ruido. Si se detecta esto último, se toma como valor máximo el más grande de los leídos hasta antes del cambio y se pasa al funcionamiento en modo de búsqueda de mínimos. Al buscar el mínimo, el comportamiento es a la inversa, se mira si el siguiente valor es menor, y solo se tiene en cuenta si es mayor que el último valor en el caso que también se supere un margen para asegurar que no ha sido una variación debida al ruido. Una vez localizado un mínimo, se pasará de nuevo a encontrar máximos. Así pues, el valor del pico será cuando el sistema haga el cambio de buscar máximo a buscar mínimo.

Como ventajas, este algoritmo proporciona protección contra el ruido configurable mediante el umbral, además de ser capaz de resolver los picos en caso de solapamiento parcial entre pulsos. Además, los cálculos a realizar son sencillos y requieren muy poca memoria. Como desventajas, la detección del instante del pico no se realiza exactamente cuando se produce, sino que se ve retrasado hasta el momento en que se detecta una caída de tensión suficientemente elevada como para poder descartar la influencia del ruido.

2.5.1.4 Procesado digital de pulsos usando la técnica de media móvil

Esta técnica de procesado de pulsos [18] se basa en calcular con cada nueva medida la media del último intervalo de tiempo T hasta el momento presente (media móvil).

Para esto se suman las últimas muestras recibidas desde el tiempo actual hasta T antes, como se puede ver en la figura 12, lo cual forma una integral discreta, y se comparan las últimas dos integrales calculadas. Para ello, se restan, y si el resultado cambia de signo significa que ha habido un extremo en el gráfico. En concreto, si se pasa de signo positivo a negativo, se encuentra un pico, y si se pasa de negativo a positivo, se encuentra un valle.

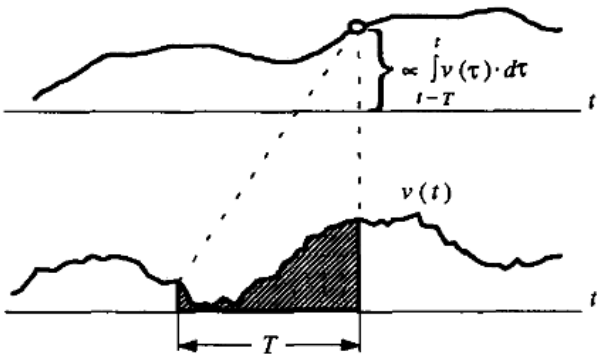

Fig. 12: Principio de funcionamiento de la técnica de la media móvil (fuente: [18])

Como el intervalo T es configurable, es posible elegir el suavizado de la señal, aumentando este valor. A más suavizado, más se tarda en encontrar el pico pero se tiene más protección contra el ruido. En cambio, si se usa un intervalo más pequeño, la detección se produce antes pero hay más probabilidades de que el ruido provoque una mala decisión.

Para que se detecten de manera adecuada los pulsos, no obstante, es necesario que se cumplan unos requisitos en cuanto a la distancia temporal entre ellos, con tal que no se solapen tanto que sean indistinguibles. Además, este algoritmo requiere de capacidad de cálculo y memoria para calcular y guardar la suma de integración.

2.5.1.5 Estimación de tiempo de llegada de pulsos basado en ratios de muestra de pulsos

Este método[19] no propone encontrar el pico del pulso, sino el instante de llegada del mismo, puesto que está orientado más a métodos de detección de blancos para radar.

El funcionamiento es guardar un ratio formado por la división entre la muestra k_a del instante t_a del tiempo de llegada y una muestra τ puntos anterior, con $\tau \geq 1$:

$$r_a = \frac{x(k_a)}{x(k_a - \tau)}$$

La ventaja de usar este umbral respecto a los otros es que, al tratarse de un ratio, carece de unidades, y por tanto es válido para cualquier amplitud que tenga el pulso de rebote. En la bibliografía se demuestra que si se calcula este mismo ratio para las muestras entrantes tiene un comportamiento monótono decreciente hasta el momento en que el pulso se vuelve constante, y por tanto solo se ha de comprobar si el ratio calculado para el punto actual es menor o igual al ratio r_a .

Este método tiene la ventaja de una gran facilidad de cálculo, además de requerir que se guarde únicamente la muestra con la que se hará la división. Como el umbral se calcula como un ratio, no importa la potencia con la que llegue la señal, siempre que sea fácilmente distinguible del ruido. Como desventaja principal, no contempla que se puedan solapar parcialmente varios blancos. Además, encontrar el punto de funcionamiento óptimo que se obtiene al encontrar el retraso τ adecuado no es trivial.

2.5.2 Conclusión

Finalmente, se decide utilizar el detector digital con umbral de ruido del apartado 2.5.1.3 puesto que es un algoritmo de implementación sencilla que requiere pocos recursos. Adicionalmente, es resistente al ruido y tiene capacidad de discernir entre varios picos pese a que los pulsos puedan solapar parcialmente. Esto es interesante en el caso que nos ocupa, puesto que puede haber ruido ambiente y puede darse el caso que se solapen parcialmente varios rebotes por la situación de los blancos.

3 Tecnologías utilizadas

Para implementar la solución es necesario tener algún tipo de soporte físico que maneje los sensores así como herramientas de programación adecuadas para poder utilizarlo. Esta implementación, como se ha comentado, tiene un objetivo, encontrar rebotes de los ultrasonidos en los blancos a localizar, lo que implica la necesidad de tener un emisor y un receptor.

El hardware debe ser suficientemente pequeño para poder ubicarlo en los escenarios a monitorizar sin interferir en el uso normal de éstos, pero lo suficientemente potente como para realizar rápidamente todos los cálculos necesarios, puesto que se espera un funcionamiento en tiempo real.

El software debe adaptarse al hardware a la hora de programar en él, así como proveer librerías para interactuar con los datos desde un PC.

En cuanto al emisor y receptor, deben tener las características apropiadas para un uso en espacios abiertos o *indoor*, según el escenario concreto con el que se esté trabajando. Según se pretenda más o menos precisión, se espere más o menos movilidad, lo ideal sería adoptar una solución específica para cada caso.

En este capítulo se hablará de las tecnologías utilizadas, tanto a nivel hardware como a nivel software, así como los diferentes emisores y receptores usados a lo largo del proyecto. En todos los casos, se comentarán los motivos de las decisiones tomadas así como las características más importantes de cada uno.

3.1 Crossbow TelosB

Para interactuar con los sensores necesarios para el proyecto se contará con un *mote* conectado al circuito que actuará como controlador y que recibirá y procesará los datos. Un *mote* es una plataforma física con memorias ROM y RAM y un microcontrolador que actuará como unidad móvil de recepción y procesado de datos.

Para este proyecto se eligió como plataforma de procesado el *mote* Crossbow TelosB[20], a día de hoy la última revisión del Telos de Crossbow, visible en la figura 13. A continuación se detallarán los motivos de la elección como plataforma de programación.

Fig. 13: Crossbow TelosB
(fuente:
http://www.xbow.com/Products/..\Products/Produc
t_images/Wireless_images/Telos_SM.jpg)

En primer lugar, se trata de una dispositivo de código libre, programable y compatible con TinyOS, uno de los sistemas operativos libres dedicados a *motes* más extendidos, lo que implica sencillez a la hora de realizar la programación. Posee una antena y un transductor de radiofrecuencia integrado y es compatible con el estándar IEEE802.15.4, lo que implica que puede comunicarse con otros *motes*, siempre que cumplan el mismo estándar. Está pensado para la experimentación, con lo que incluye un puerto USB para conectar al ordenador y poder programar o interactuar con el funcionamiento del *mote*, lo que permite la recogida de datos en línea para efectuar todo tipo de operaciones, incluyendo centralizado de datos, compensación de errores y un largo etcétera.

Especificaciones	***TPR2400CA***	
Módulo		
Procesador	16-bit RISC	
Memoria Flash Programable	48K bytes	
Memoria Flash Serie	1024K bytes	
RAM	10K bytes	
EEPROM	16K bytes	
Conversión Analógica-Digital	12 bit ADC	
Conversión Digital-Analógica	12 bit DAC	
Consumo	1.8 mA	Modo activo
	5.1 µA	Modo *sleep*
Transductor RF		
Banda de frecuencias	2400 MHz – 2483.5 MHz	
Velocidad de transmisión	250 kbps	
Potencia RF	-24 dBm a 0 dBm	
Sensibilidad de recepción	-90 dBm a -94 dBm	
Rechazo de canal adyacente	47 dB	> 5MHz separación
	38 dB	< 5MHz separación
Rango *outdoor*	75 m a 100 m	
Rango *indoor*	20 m a 30 m	
Consumo	23 mA	Modo recepción
	21 µA	Modo pausa
	1 µA	Modo *sleep*
Electromecánica		
Batería	2x AA	
Interfaz de usuario	USB (v1.1 o superior)	
Tamaño (mm)	65 x 31 x 6	Sin baterías
Peso	23 g	Sin baterías

Tabla 1: Características técnicas del Crossbow TelosB

Además, es habitual en entornos de desarrollo, y esta escuela ya estaba dotada de varios Crossbow TelosB de antemano, que había utilizado satisfactoriamente en proyectos anteriores.

La característica más relevante de la tabla 1 a la hora de programar el *mote* será la limitación de memoria (flash, RAM y EEPROM), que condicionará el tamaño máximo tanto del programa en sí como del espacio que pueden ocupar las variables. A lo largo del desarrollo de la aplicación se tendrán en cuenta estos valores puesto que sobrepasarlos provoca un funcionamiento incorrecto.

Para comunicarse con los sensores, y realizar las medidas, además de varios puertos el Crossbow TelosB está dotado de un ADC y un DAC, ambos de 12 bits. El primero son las siglas de Analog-to-Digital Converter, conversor analógico-digital, que podrá realizar una lectura analógica del voltaje en la entrada y transformarlo a valores computables. El segundo es un Digital-to-Analog Converter, conversor digital-analógico, que servirá para justo lo contrario, es decir, convertir un valor interno de la memoria o programación en un valor analógico de tensión que podría conectarse, por ejemplo, como entrada a un multímetro.

Hay otras características importantes en la tabla 1 que sustentan la decisión. Un alcance de la radio 20 metros a 30 metros en entornos cerrados es más que suficiente para cubrir una habitación a monitorizar, y los 75 metros a 100 metros en entornos abiertos son adecuados en caso que se utilice en entornos móviles. En ambos casos el radio de acción permite crear una red para comunicar los nodos entre ellos, aunque esto queda fuera del alcance de este proyecto. Por otra parte, las dimensiones y el peso son suficientemente pequeños para no interferir con el entorno, y el tipo de baterías que utiliza, dos pilas AA, son comunes y económicas. Además, está especialmente preparado para operaciones de bajo consumo con tal de alargar la vida útil de la batería al estar capacitado para entrar en modo de espera.

3.2 TinyOS

Elegir el sistema operativo (en inglés Operating System, OS) con que dotar a los *motes* a utilizar es una elección importante, y más en un sistema ad-hoc. Cuando los mismos nodos son los que deben procesar gran parte de la información que se recibe, se necesita de herramientas que ayuden a la programación e implementación de los algoritmos de localización.

En este caso se ha decidido utilizar TinyOS para la programación de los TelosB, en primer lugar porque es totalmente compatible con el hardware, y en segundo lugar porque al tratarse de un sistema operativo de uso bastante extendido en el ámbito de los *motes*, se tiene una gran colección de librerías, ejemplos e implementaciones alternativas muy útiles a la hora de programar el sistema.

TinyOS es un sistema operativo libre de código abierto diseñado para redes de sensores *wireless* embebidos, con un especial énfasis en la reducción del tamaño del código compilado para funcionar en los *motes* más sencillos manteniendo todas las opciones para los *motes* más potentes[21]. Al tratarse de un OS libre, cualquier usuario puede editar y reutilizar el código fuente, de manera que la comunidad de usuarios también alimenta directamente el desarrollo de nuevas versiones de TinyOS con sus parches y modificaciones.

3.2.1 Funcionamiento de TinyOS

3.2.1.1 nesC

TinyOS está programado en nesC[22] (network embedded system C), una variante de C orientada específicamente a las arquitecturas de hardware de sensores.

La programación en nesC está dividida en componentes software, que de alguna manera son un paralelo con los componentes hardware de los sensores. Estos componentes son de dos tipos, módulos y configuraciones.

Los módulos son la parte que implementa el sistema, los que poseen la lógica y el código que constituyen el programa que se carga en los *motes*. Estos módulos están divididos en definición de interfaces utilizados e implementación. Los interfaces utilizados se usan en la implementación del módulo (ej. una interfaz para interactuar con los leds) para proveer funciones para la interactividad entre el código y los componentes hardware del sensor. Así, se tienen comandos (*commands*), que son llamados por componentes de más alto nivel, y eventos (*events*), que señalizan acciones a componentes de más alto nivel, generalmente la finalización de un comando. Estos eventos se lanzan como interrupciones, deteniendo el flujo del programa, para atender con el máximo de celeridad posible, pese a no tener capacidades de tiempo real. También se tienen tareas (*tasks*), que son atómicas entre sí, es decir, que no pueden interrumpirse unas a otras. En cambio, estas *tasks* sí pueden ser interrumpidas por señales procedentes de eventos, cosa que se ha de tener en cuenta en la programación.

Por ejemplo, en caso de que se interactúe con una interfaz para enviar paquetes, primero se llama al comando send() de la interfaz, que al terminar su ejecución señaliza al evento sendDone() cuando termine. En este caso, mientras que send() está implementado según el sensor que se utilice, puesto que la comunicación con la radio depende del software disponible, el evento sendDone() tiene que ser programado para responder a las necesidades del usuario. En el caso de las tareas, en cambio, lo que sucede es que se preparan en la pila de ejecución y se comienzan a procesar cuando hay espacio libre en el procesador.

Las configuraciones se encargan de hacer las conexiones virtuales (*wiring*) entre los diferentes módulos así como de realizar la instanciación de los componentes genéricos. De alguna manera, lo

primero representa una conexión de hardware en la cual una salida de un módulo se conecta a la entrada de otro o viceversa, solo que en el caso del software las conexiones tienen un sentido definido, y por tanto es posible tener configuraciones que no son posibles en el caso de unión por cables en un soporte físico. Lo segundo, los módulos o configuraciones genéricos, son piezas reutilizables de código ejecutable y de conjuntos de relaciones, respectivamente, que sirven para aprovechar varias veces un patrón de conexiones.

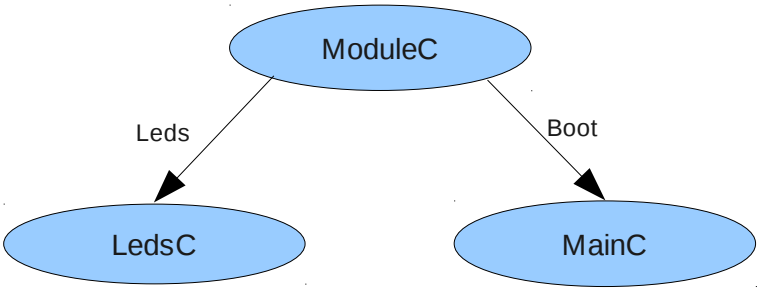

Fig. 14: Esquema de conexiones de ejemplo

Un ejemplo posible es realizar un programa muy sencillo que al iniciarse encienda un led de la placa. Suponiendo que el módulo que contiene el ejemplo se llame ModuleC, hay que conectarle el módulo MainC, que contiene Boot y se encarga de realizar operaciones al iniciar el nodo, y el módulo que interacciona con los leds, LedsC. Tanto LedsC como MainC son *front-ends* que varian según la plataforma destino, y se interconectan para realizar una implementación especificada dentro de ModuleC. La conexión se realizaría:

```
ModuleC.Leds -> LedsC
ModuleC.Boot -> MainC
```

La conexión final quedaría como la representada en la figura 14 y el programa estaría listo para un funcionamiento sencillo, como por ejemplo hacer que se encendiera un led al inicio.

Esta necesidad de especificación de interconexión entre las diferentes partes del programa global a cargar es lo que más diferencia nesC de C o similares.

3.2.1.2 Propiedades de TinyOS

Al tratarse de una programación específica para plataformas de red embebidas, como el caso de TelosB, trata de atacar dos puntos importantes. Por una parte intenta reducir al máximo el peso tanto en la memoria RAM como la ROM del dispositivo donde instalarlo. Por otra parte, usa técnicas para reducir el consumo del procesado, con tal de alargar la vida de la batería.

Para lo primero, evitar una ocupación elevada en la memoria, no posee ningún tipo de interfaz gráfica; para recuperar los datos de manera visual es necesaria la conexión de un *mote* con un PC así como una programación adecuada en ambos. Además, no se tiene un procesado multi-hilo habitual en los sistemas operativos de hoy en día, sino que solo se cuenta con una pila de memoria donde almacenar los datos. Aún y así, permite que las *tasks* sean interrumpidas por eventos, aunque estos no pueden interrumpir a otros eventos, de manera que no se posee respuesta en tiempo real auténtica. En todo caso, para aumentar la capacidad de reacción del sistema, se puede hacer que las llamadas a eventos acaben de realizar los cálculos en tareas, de manera que dejan libre al sistema para recibir nuevas señales. Por otra parte, es posible utilizar la especificación *async*, lo que provoca que se ejecutara el código de forma asíncrona. Este código interrumpe cualquier otro código que se esté ejecutando en ese momento, incluyendo otro código asíncrono, con lo cual se puede utilizar para aplicaciones que requieran un *timing* crítico, como por ejemplo la recepción radio. Los eventos asíncronos si pueden interrumpir a otros eventos. También como parte del ahorro de memoria, no existe diferencia entre memoria de sistema y memoria de usuario, con lo cual es importante tener en

todo momento bajo control los punteros y limitar su uso a cuando sea estrictamente necesario, puesto que un puntero descontrolado puede escribir en memoria de sistema y volverlo inestable o incluso dejarlo inservible hasta una nueva instalación.

Para lo segundo, la reducción del consumo, se impide el uso de *polling*, que requiere interrogar a los diferentes componentes físicos para detectar su ocupación, y opta por la utilización exclusiva de interrupciones para la respuesta del sistema. Además, utiliza un modelo *split-phase* (fase partida), lo que quiere decir que, como se ha visto en el apartado de nesC, se llama a un comando que realiza una acción (p ej. iniciar una cuenta atrás de 100 ms) y, si el sistema no tiene nada pendiente, entra en estado *sleep* para minimizar el consumo. Una vez se completa la acción (p ej. han pasado los 100 ms), se lanza una señal que activa un evento que se ejecuta, de manera que está despierto solamente cuando sea necesario. Adicionalmente, en la versión de TinyOS 2.0 se incorporó en el núcleo un sistema de control de energía[23] que crea lo que se denominan *power locks*, que consisten en bloquear el acceso a los servicios del sensor para aumentar la eficiencia energética del sistema. Con esto se tiene entre un 98.4% y un 99.9% de la eficiencia energética de un programado a mano específico para una aplicación, solo que con este sistema no se requiere de instrucciones de código especiales, puesto que se encarga de manejar los consumos automáticamente.

Con estas decisiones de diseño, por tanto, se adapta a las necesidades y limitaciones de un sensor con funcionamiento en red, de manera que se aprovecha de manera adecuada la memoria disponible y se alarga la vida útil de la batería.

3.3 Emisor y receptor de ultrasonidos

En este apartado se comentarán las principales características de los diferentes emisores y receptores de ultrasonidos utilizados a lo largo del proyecto. Concretamente, se hablará de los transductores en sí, usando una circuitería que no implique una variación en el funcionamiento.

En primer lugar se analizará un transductor con un circuito propio, tanto para emisión como para recepción, así que es interesante conocer las características que resultarían, obtener sus características para ver si se adaptan al proyecto. Una opción a tomar en cuenta consiste en colocar sobre el emisor un cono de plástico realizado en un torno (ver figura adjunta 15), con la idea de hacer al emisor omnidireccional. Esto evitaría tener que poner varios emisores puesto que con uno solo se cubrirían todas las direcciones, interesante en el caso que los blancos pudieran estar dotados de receptores de ultrasonidos. Para comparar el funcionamiento con y sin cono se realizan pruebas de distancia máxima y se obtiene el diagrama de radiación para ambos casos.

Fig. 15: Montaje utilizando el cono torneado

En segundo lugar se explicarán las características del emisor-receptor comercial SRF02, incluyendo cómo se puede interaccionar con él vía software. Esto es muy importante a la hora de programar los nodos.

3.3.1 Alcance del emisor propio

Las pruebas de alcance consisten en utilizar un TelosB que está dotado de un receptor de ultrasonidos diferente al del proyecto actual. Este dispositivo tiene una programación que indica mediante la interacción con los leds si se detectan o no ultrasonidos.

El procedimiento para medir el alcance es el siguiente: el emisor se mantiene estático, por estar montado en la *protoboard*, y se va alejando progresivamente el receptor hasta que los leds se mantengan constantemente encendidos, lo que significa que la señal ya no puede ser detectada. Como se quiere medir el alcance máximo, el emisor y el receptor están encarados, o bien entre ellos

para calcular el rango de visión directa, o bien hacia el punto de rebote en la pared, en el caso del cálculo del rango de visión indirecta con un rebote.

3.3.1.1 Prototipo directivo

Para este caso se realizan medidas alimentando el emisor a 18 V y a 9 V, para ver los cambios que se producen al variar la alimentación.

Los resultados obtenidos se ilustran en la tabla 2:

Tensión de alimentación	**Alcance**	
	Con rebote	*En visión directa*
18 V	12 m	22 m
9 V	8 m	20 m

Tabla 2: Medida de los alcances del prototipo de emisor directivo

Como puede comprobarse, al reducir la tensión de alimentación del transductor de ultrasonidos se reduce también el alcance máximo con el que puede recibirse la señal. Como es de esperar, en el caso de producirse un rebote el alcance es menor que cuando se tiene visión directa, debido a que parte de la energía del pulso de ultrasonidos se disipa en la pared.

3.3.1.2 Prototipo omnidireccional

En este caso no se enfoca el extremo superior del emisor con el receptor, sino que como teóricamente debe comportarse de manera tal que distribuya la potencia paralelamente a la superficie del transductor, se pone al emisor apuntando hacia el cielo. Los resultados están en la tabla 3.

Tensión de alimentación	**Alcance**	
	Con rebote	*En visión directa*
18 V	7 m	14 m
9 V	4 m	7 m

Tabla 3: Medida de los alcances del prototipo omnidireccional

Como es de esperar, en este caso se obtienen alcances menores ya que se ha de distribuir la potencia de la señal en un volumen mayor que para el caso anterior. Para medir los alcances con rebote aquí se pone de manera más perpendicular posible el receptor con respecto a la pared, para evitar que haya visión directa.

3.3.2 Diagrama de radiación del emisor y receptor propios

Una vez determinado el alcance máximo en estos escenarios, el siguiente paso para caracterizar al emisor es encontrar su diagrama de radiación. Para ello se utiliza el prototipo del receptor para realizar la lectura de la señal de llegada. Éste está conectado al osciloscopio, con una escala temporal de 500 ms/div y una escala vertical de 200ms/div. Con esto se puede llenar una tabla con el valor máximo de tensión de pico recibido en los últimos 5 s.

Para esta prueba, el receptor y el emisor se mantienen a una distancia constante, de manera que no haya variaciones en la tensión medida debidas al movimiento. Una vez situado el emisor, haciendo uso de un transportador de ángulos se va variando el ángulo de incidencia del transductor de

ultrasonidos sobre el receptor, para obtener las diferentes medidas de tensión. Para minimizar errores, en cada posición se tomaron varias medidas, y se eligen las 3 más repetidas para realizar la media. Con esto se puede descartar el efecto de rebotes pasajeros o imprecisiones en el ángulo debido, por ejemplo, al viento que mueva al emisor o el receptor.

Como el modelo de transductor de ultrasonidos usado como emisor y receptor es equivalente, el diagrama de radiación del prototipo directivo del emisor será válido también para el receptor.

3.3.2.1 Prototipo directivo

Para este caso no se miden los ángulos mayores que 90º y menores que 270º, debido a que las lecturas serían falseadas por la presencia de una pared situada unos metros detrás del emisor. El resto de ángulos, es decir, el semicírculo frontal, son medidos en intervalos de 10º, realizando posteriormente medidas adicionales en los 15º, 25º, 335º y 345º. Una vez realizadas todas las medidas, se calcula la media para cada ángulo, se normalizan los valores dividiendo por el máximo y se pasa a factor de potencia elevando este coeficiente al cuadrado.

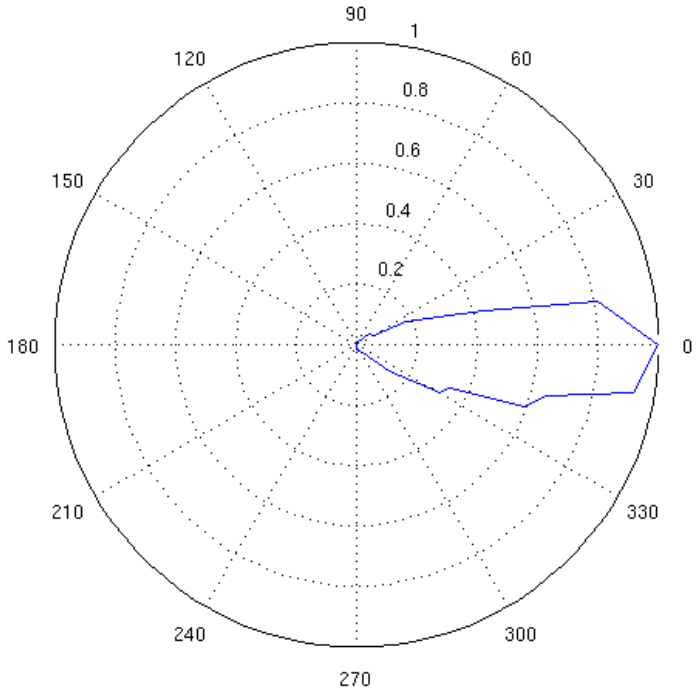

Fig. 16: Diagrama de radiación del prototipo del emisor directivo

Con estos valores, en Matlab se realiza el gráfico de la figura 16 en coordenadas polares.

Como puede verse, se trata de un único lóbulo estrecho, con un ancho de haz resultante de aproximadamente unos 30º. Como el transductor es un dispositivo circular, y por tanto, simétrico, su comportamiento para el caso de elevación será aproximadamente igual que en el caso mostrado, así que la directividad es de aproximadamente:

$$D = \frac{4\pi}{\Delta\theta_1 \cdot \Delta\theta_2} \approx \frac{4\pi}{\pi/6 \cdot \pi/6} = 45.8366 = 16.612\,dB$$

3.3.2.2 Prototipo omnidireccional

Para este caso es interesante caracterizar el comportamiento en los tres ejes, así que se realizan dos diagramas de radiación diferentes.

El primero es medido paralelamente a la superficie del transductor, realizado en los 360º de 10º en 10º. Se obtiene el gráfico de la figura 17.

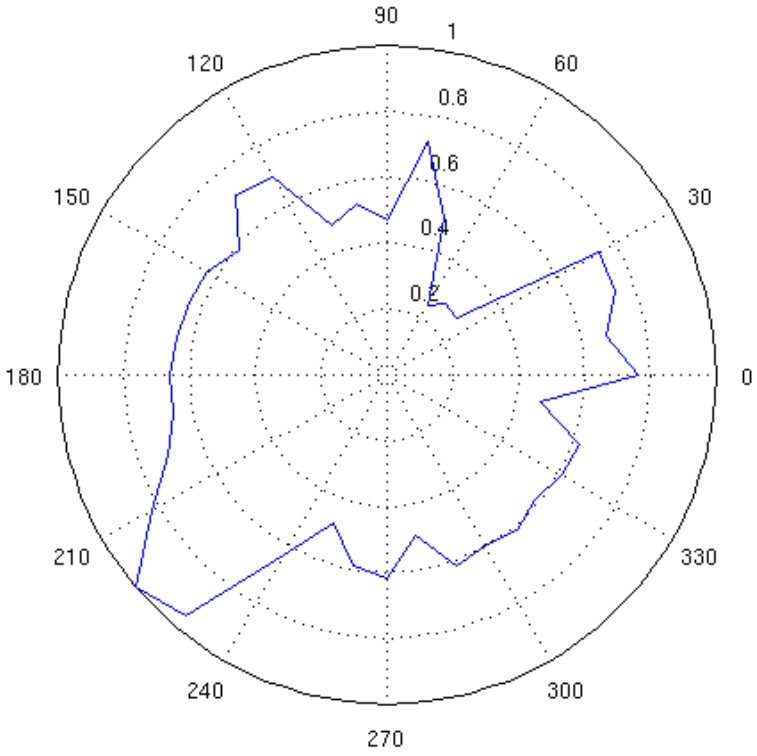

Fig. 17: Diagrama de radiación paralelo a la superficie del cono

De éste se esperaba que fuera omnidireccional, es decir, que enviara la misma potencia en todas direcciones. Aunque no es exactamente omnidireccional, tiene un margen de error aceptable. El lóbulo ubicado entre 210º y 240º así como el valle entre 30º y 60º corresponden a una pequeña desviación del cono situado sobre el emisor que puede ser corregido modificándolo.

El segundo diagrama a realizar se hace colocando el cono de manera que apunte directamente al receptor, y esto se tomó como 0º. A partir de ahí, se hace rotar en el eje central del emisor hasta los 180º, de manera que el cono apunte justo en la dirección opuesta. Como se ha visto con el diagrama anterior que el cono no está centrado, para cada ángulo en el otro eje en que se calcula el diagrama resultarían unos valores distintos, así que se decide tomar como simétrica esta medida de media circunferencia.

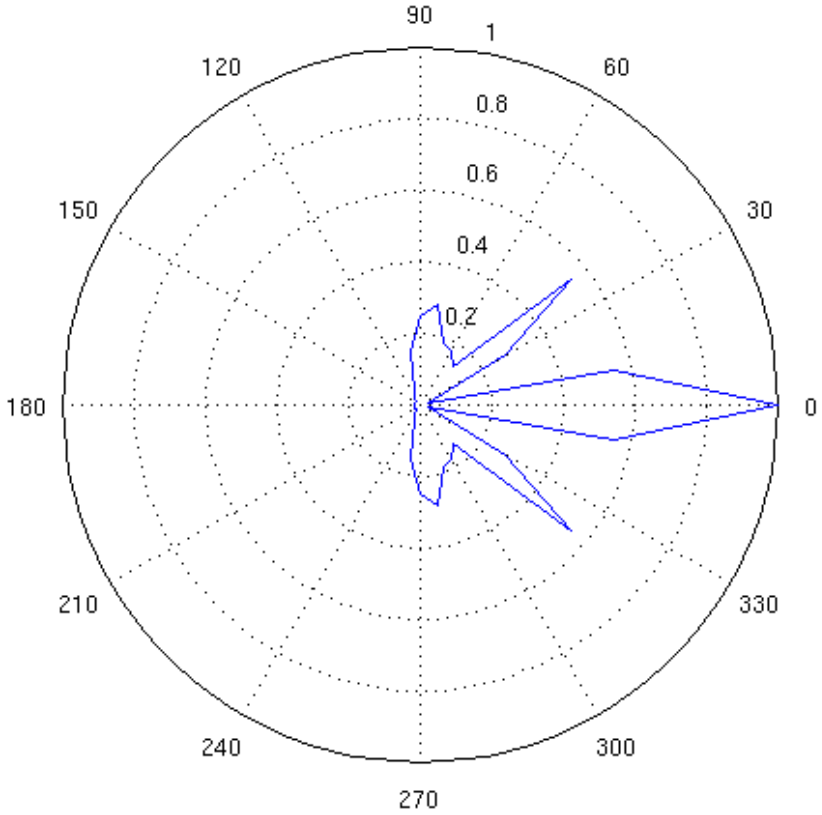

Fig. 18: Diagrama de radiación perpendicular a la superficie del emisor

Idealmente, el diagrama de la figura 18 debería tener un cero en la dirección del cono y un único lóbulo en un ángulo de aproximadamente 90º, perpendicular a la superficie del transductor. Como puede verse, no se cumple ninguno de los dos requisitos. En primer lugar, en la dirección del cono, 0º, se tiene un máximo. Se realizan pruebas variando el ángulo de incidencia y tapando la salida lateral para comprobar que no sea a causa de algún tipo de rebote que alcance justamente al receptor, pero aún y variando la posición drásticamente, la única manera de reducir este máximo es tapando directamente delante del cono del emisor. Esto deja la conclusión que el cono es permeable para una emisión directa de ultrasonidos. Pese a ello, en caso de usarse un sistema similar esta potencia puede enfocarse en dirección al cielo, por lo que es de esperar que no interfiera excesivamente en la detección. En segundo lugar, se puede comprobar que aproximadamente el comportamiento perpendicular a la salida del cono es como el deseable, pese a tener demasiada potencia enfocada hacia los ángulos cercanos a 30º. Esto puede intentar corregirse variando el tamaño y el ángulo del cono, para que el comportamiento se pareciera más al esperado.

Así, mejorando el montaje y los materiales sería posible conseguir el efecto deseado y, por tanto, cubrir más ángulo con un solo emisor, en caso que se quisiera dotar a los blancos de receptores propios.

3.3.3 SRF02

El SRF02[24], mostrado en la figura 19, es un sensor de distancias por ultrasonidos capaz de detectar un único rebote a una distancia máxima de 6 m. Como posee un solo transductor, la distancia mínima para la detección es de 15 cm, que corresponde al tiempo que tarda en cambiar de modo de funcionamiento: primero emite un pulso de ultrasonidos a 40 KHz con una duración de 8 ciclos de reloj y después cambia a receptor y escucha el primer eco, que procesa y devuelve esta medida en centímetros, pulgadas o microsegundos. Su diagrama de radiación, visible en la figura 20, es directivo, aunque menos que en el caso del emisor propio.

Fig. 19: Dispositivo SRF02

Para este proyecto era interesante la opción que posee de emitir un pulso de ultrasonidos sin esperar respuesta, puesto que permite recuperar los datos mediante un receptor propio. Con esto se pasaba a un funcionamiento multirrebote, en el cual un solo pulso podía devolver información de uno o más blancos.

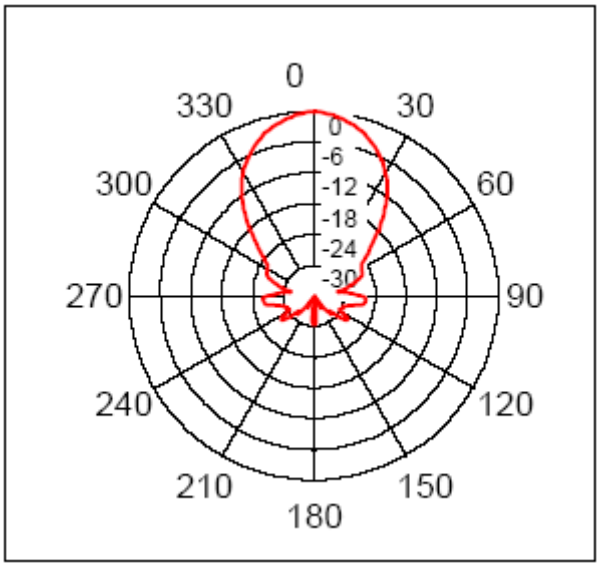

Fig. 20: Diagrama de radiación de un SRF02 (fuente: http://www.acroname.com/robotics/parts/R287-SRF02a.jpg)

Este sensor se puede conectar mediante puerto serie o interfaz I2C al TelosB. El puerto serie es un interfaz físico donde la información se envía bit a bit a través de la UART (Universal Asynchronous Receiver-Transmitter), cuyo homónimo en TinyOS comparte recursos con otros elementos del *mote*.

El interfaz I2C (Inter-Integrated Circuit) es un estándar de comunicación para conectar con periféricos a baja velocidad.

En primer término sólo se dispone de los drivers programados para el modo de funcionamiento en puerto serie, pero esto presenta una serie de problemas. Por ejemplo, al utilizar funciones para imprimir datos por pantalla se obtienen errores de funcionamiento, y se limita el uso de la radio. Por ello, se pasa a la programación de los drivers para el interfaz de I2C, que simplifican la interacción con el resto de componentes del sistema.

Hay que prestar especial atención a que no se pueden leer los resultados de manera inmediata, sino que, según la documentación, es necesario esperar al menos 66 ms. Para ello, se hace en primer lugar una función que sirva para enviar el comando al dispositivo y otra que permita recuperar los datos más adelante. No se utiliza el modelo *split-phase* de TinyOS de forma global en toda esta interacción para tener control total sobre el momento en que se realiza la encuesta al dispositivo. Para realizar esta lectura se crean varias funciones para leer de los diferentes registros del SRF02.

A la hora de la programación con el resto del código se debe tener en cuenta que el interfaz I2C comparte recursos con la radio, y por tanto es necesario desactivarla en el momento de enviar las órdenes al SRF02 y volver a encenderla una vez posteriormente. El uso de este interfaz sobre el puerto serie también permite utilizar escritura en pantalla desde el nodo, para poder examinar detalladamente el funcionamiento interno durante la programación.

4 Implementación de un sistema de localización

El objetivo principal de la implementación es realizar una solución adaptable a varios escenarios, concretamente dos casos: la localización de objetos en entornos exteriores móviles, y la localización de personas en una sala.

Para el primer caso lo mejor es que el emisor y el receptor se encuentren en nodos separados, de manera que se comuniquen entre ellos por radiofrecuencia antes de emitir el pulso.

En este caso el nodo dotado del emisor manda un paquete radio indicando al receptor que debe activar la escucha por ultrasonidos. Éste, durante un intervalo de tiempo determinado, recibe datos de ultrasonidos para posteriormente computar la señal recibida y decidir a qué distancia se encuentran el uno del otro. Es posible enviar este paquete radio sin problemas de temporización debido a que el tiempo de propagación del ultrasonido es muy inferior al de la luz.

En el segundo caso, es impensable dotar a cada persona de un dispositivo, ya que se pretende que sea utilizable en entornos en los que no tiene sentido pedir a los objetivos la posesión de ningún elemento, como por ejemplo en una tienda.

En este caso la idea es emitir un pulso de ultrasonidos contra el espacio a observar y esperar a leer los rebotes producidos por los blancos y los obstáculos que se puedan encontrar en esta zona, de manera similar a un Radar, como se simplifica en la figura 21.

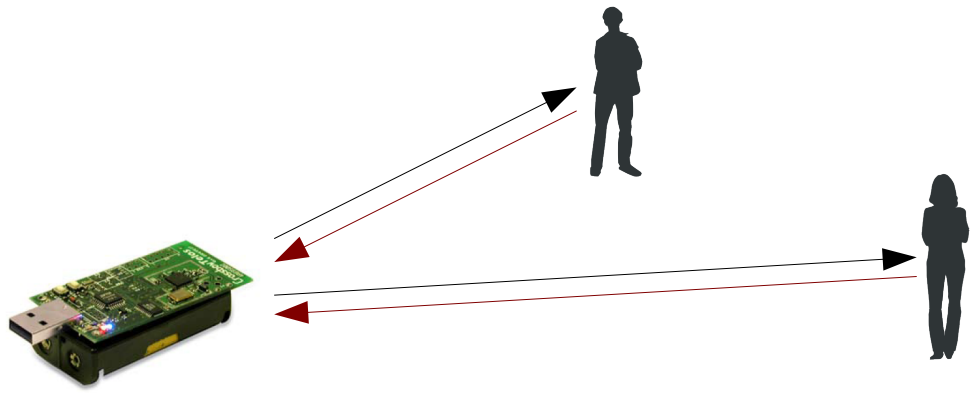

Fig. 21: Esquema de funcionamiento del sistema

El circuito transmisor planificado que utiliza el emisor comentado en el apartado 3.3.1 está controlado por una entrada, que va conectada a un pin del *mote*, que activa la transmisión de ultrasonidos únicamente mientras ésta se encuentre en nivel alto. Si se usa como transmisor el SRF02 del apartado 3.3.3 , en cambio, se debe apagar la radio antes de utilizarlo puesto que comparten recursos, y se interacciona mediante los drivers creados. El receptor que se pretende utilizar en ambos casos pone en su salida la envolvente de la señal recibida amplificada, de manera que el pin conectado a esta salida debe muestrear el canal cuando se de la orden, valorar estos datos y emitir un veredicto sobre las distancias a las que encuentra rebotes.

Posteriormente, estos datos con los rebotes se envían vía radio a un nodo central conectado a un PC que mostrará los resultados por pantalla. El procesado conjunto de todos los datos en el ordenador para posicionar totalmente los blancos y eliminar los rebotes producidos por elementos estáticos del escenario (como sillas o mesas) queda fuera del alcance de este proyecto.

4.1 Familiarización con el entorno de trabajo

Antes de comenzar a programar en el nodo hay que preparar el entorno de trabajo, tanto a nivel de hardware como de software.

Para lo primero se realiza un programa muy básico de lectura de tensión para mostrarlo por pantalla o bien desde el mismo nodo que toma las medidas o bien desde un nodo que las reciba por radio.

Para lo segundo, se crea un entorno de trabajo portable mediante una máquina virtual dotada de todas las aplicaciones necesarias para el diseño e implementación del software.

4.1.1 Programación inicial en el nodo

Como primera toma de contacto con el TelosB y las herramientas de programación se comienza con un programa prototipo que lee la tensión en uno de los pines de entrada del *mote* y la muestra por pantalla mediante el uso de *printf*, estando conectado directamente al ordenador mediante el USB. Para esto se modifica el límite de caracteres a mostrar por pantalla con tal de poder escribir toda la información, así que se altera este valor en el archivo que contiene las opciones de compilación, el *Makefile*.

A modo de resumen, el funcionamiento interno del siguiente paso en el programa consiste en guardar en un vector las muestras tomadas en un mensaje, y enviarlo por radio cuando se llene. Esto se repite hasta que se considera que se tiene un número total de muestras enviadas adecuado, momento en el que se desactiva el receptor. Para enviar el mínimo de paquetes radio, se desea que éste sea lo más grande posible, así que se modifica la opción en *Makefile* para que tenga el tamaño máximo permitido por el sistema operativo, 112 bytes.

Antes de programar el muestreo por el puerto conversor analógico-digital (ADC), se necesitan tomar una serie de decisiones. En primer lugar, se ha da considerar cuántos canales utilizar, puesto que es posible leer de varios a la vez. En segundo lugar, se ha de elegir el método por el que tomar las muestras:

- Una única muestra (se lanza una única interrupción)
- Una serie de muestras tomadas de una en una (cada muestra nueva lanza una interrupción)
- Una serie de N muestras (se lanza una única interrupción)
- Una serie de muestras tomadas de N en N (cada serie de N muestras lanza una única interrupción), con una N comprendida entre 1 y 16, ambos inclusive.

A la hora de considerar el número de canales, como sólo se necesita leer los datos de un único transductor de ultrasonidos, se utiliza la recepción por un sólo canal, el ADC0. Debido a que se toma un número muy superior a 16 muestras cada vez que se activa la lectura de los ultrasonidos, el método para tomar las muestras que se utiliza es la cuarta opción, con N por valor de 16. Esto permite reducir al máximo posible el número de interrupciones que manda el procesador, siendo por tanto el más eficiente.

Al pasar los datos de la lectura del ADC al mensaje radio se ha de tener en cuenta que el mensaje radio y el vector donde se guardaban las muestras tenían tipos diferentes. El tipo de datos guardado en el mensaje es *nx_uint_16t*, un *unsigned int* de un tipo especial compatible en todas las plataformas que pueden recibir este paquete, al tener una ordenación específica de los bits para evitar los problemas de comunicación entre dispositivos *big endian* y *little endian*. Esta conversión de tipo no se puede realizar copiando directamente la memoria apuntada por el vector de las muestras al puntero que representaba el payload de datos del paquete, puesto que cada uno tenía una ordenación distinta, si no que se ha de realizar una asignación muestra a muestra para que se

convierta internamente de la manera adecuada.

Una vez tomadas las medidas se envían haciendo un *broadcast* vía radio con tal de poder leerlas en un PC. Para ello, se construye un tipo de paquete radio donde poner los datos, cuya distribución es también visible en la figura 22, que contiene 2 bytes para señalizar el nodo origen, 1 byte de *flags* de control y 96 bytes de información con las medidas. Este valor se elige así recordando que el tamaño total del paquete radio no puede superar 112 bytes por limitaciones del sistema operativo, pero además porque resulta en un total de 48 muestras, que es múltiplo de 16, que hace más sencillo utilizar el muestreo más eficiente posible. Aunque cada muestra ocupa 2 bytes, el conversor analógico-digital del TelosB es de 12 bits, pero se decide que en lugar de despreciar los 4 últimos bits para tener muestras de 1 byte es mejor tomar el valor completo para las primeras pruebas. En apartados posteriores se explicará como se varía esto para descartar los 4 bits menos significativos para reducir el número de mensajes a enviar.

Fig. 22: Distribución del paquete radio

Para dejar suficiente margen para ver cualquier tipo de rebote en la señal de retorno, como el pulso utilizado era de unos $200\,\mu s$, se utilizaba un período de sampleo de $160\,\mu s$, y una captura de un total de 576 muestras cada vez, lo que provocaba una escucha durante 92.16 ms. Se elige este número de muestras por ser múltiplo de 48, para poder mantener la configuración de lectura del ADC.

Una vez el sistema de radio funcionaba, la manera de recibir los datos en el ordenador consiste en tener un nodo conectado por USB que tenga instalado el programa *BaseStation*, disponible en el núcleo de TinyOS 2.x. Este código redirige los mensajes radio en el mismo formato que llegan, es decir, como en la figura 22, así que ejecutando desde el PC la aplicación Java *MsgReader*, también proveniente del código fuente de TinyOS, se pueden ver por pantalla los mismos mensajes que han sido enviados por radio, para analizarlos. No obstante, al leer los datos en el PC resultó que algunos mensajes se perdían debido a que la radio debía enviar un paquete y, antes de poder liberarla, ya se debía mandar el siguiente.

Para corregir esto, se añaden temporalmente tres archivos más al programa, *RadioAppC.nc*, *RadioC.nc*, y *Radio.h*, que llevaban la configuración, el código y las constantes del módulo de radio respectivamente. Estos archivos hacen de capa intermedia para interactuar con la radio de manera transparente en el programa principal, y se encargan de implementar un sistema de cola que guarda los paquetes y retrasa su envío hasta tener disponibles los recursos.

Antes de montar el circuito que lleva el transductor, se pasa a comprobar el correcto funcionamiento del sistema leyendo tensión de referencia y tierra. El siguiente paso es leer del transductor de ultrasonidos, pero para tener los datos de manera más visual se necesita de programación específica en el PC.

4.2 Herramientas de visualización en el PC

No siempre es posible tener conectado directamente con USB todos los nodos que se quieran monitorizar, pero el uso de un único *BaseStation* que sí esté conectado permite ver todas las transmisiones radio del proyecto. No obstante, las herramientas por defecto de TinyOS no se adaptan completamente a las necesidades, así que se programan unas específicas.

En el ordenador de trabajo se tiene instalado Ubuntu Linux[25], en el cual se tiene una máquina virtual con VMWare[26], a la cual se le instala XubunTOS[27], distribución de Linux basada en Ubuntu que trae una instalación completa de TinyOS 1.x y TinyOS 2.1. Desde este máquina se realiza, mediante el procesador de textos Vim[28] y el entorno de programación NetBeans[29], el desarrollo de todo el proyecto. Este último es un programa de código abierto para editar y manejar proyectos en Java respaldado por Sun Microsystems. Posee la posibilidad de editar JFrames (ventanas Java) de manera mucho más sencilla y visual que simplemente utilizando la programación en modo texto. Con este programa se puede editar utilizando únicamente el ratón la posición, tamaño y demás propiedades visuales de las ventanas, de manera que genera automáticamente el código en modo texto necesario para los ajustes pedidos. También cuenta con muchas otras utilidades interesantes, como la integración global de todos los componentes de un proyecto, de manera que buscar cualquier ítem existente en él es el más fácil. Además cuenta con debugger, para el seguimiento detallado de variables en tiempo de ejecución, y profiler, para monitorizar en todo momento el consumo de memoria y CPU del proyecto. Estas funciones hacen mucho más sencilla la depuración del código.

Se elige programar la interacción con los paquetes que llegan por el puerto USB en Java, en primer lugar porque es uno de los lenguajes soportados por los creadores de TinyOS, que incorpora librerías para interactuar con los *motes*, y en segundo lugar porque es multiplataforma, con lo cual cualquier ordenador, sin importar el sistema operativo que lleve, puede ejecutar el código de interpretación de los datos. Además, Java cuenta con una gran colección de librerías y herramientas de muy alto nivel que facilitan la programación gráfica.

El primer paso para analizar los datos obtenidos es interpretarlos. Al arrancar el programa se designa un puerto de escucha para leer todos los datos que se recibían vía radio en el *BaseStation*. Así se conseguía ver el mensaje bit a bit en el PC, pero lo interesante era poder interactuar con estos datos. Para ello, TinyOS incorpora una utilidad, *mig*, que compila los tipos de mensajes de nesC en Java: crea un archivo *.java* para cada tipo de mensaje, y los provee de instrucciones para extraer o insertar información. Teniendo los tipos de mensajes en Java ya se pueden insertar como un archivo más en el proyecto de NetBeans para ser compilados y utilizados como cualquier otra clase, simplemente incluyéndolos como librerías.

La naturaleza modular de Java permite añadir y quitar elementos según se necesitan sin necesidad de modificar los que ya no sean utilizados. Con esto ya se pueden analizar todos los datos que lleguen mediante la radio, en primer término para hacer gráficos, y posteriormente para procesar los datos de los picos obtenidos.

4.2.1 Interfaz gráfico

Comprobar el correcto funcionamiento del nodo y la programación es más sencillo y visual si se puede obtener una representación gráfica de la señal recibida, para poderla comparar con la esperada.

Para ello se utiliza la librería especializada JFreeChart[30], capaz de realizar gráficos con multitud de opciones y configuraciones distintas. Se puede integrar en Swing, la librería gráfica de Java más extendida, y permite realizar gráficos de todo tipo, desde histogramas hasta gráficos circulares, pasando por diagramas de dispersión. Los datos de estos gráficos se agrupan en *datasets*, que cuentan con la ventaja que lleva integrado un sistema de actualización automática de los gráficos, de manera que si se detecta un cambio en los datos sin necesidad de señalizar nada más se produce el cambio necesario en el gráfico. También integra funciones de zoom, cursor, leyenda y un largo etcétera.

Para poder hacer uso de la librería antes mencionada se necesitaba crear un *java Panel* que sería el

contenedor del gráfico. No obstante, al usar JFreeChart se ocupa todo el panel, de manera que para añadir más funciones no se pueden añadir en este mismo *Panel*. Por ello se opta por hacer un panel externo que contenga como un objeto más otro panel interno, siendo este segundo asignado como objetivo del gráfico, como puede verse en la figura 23.

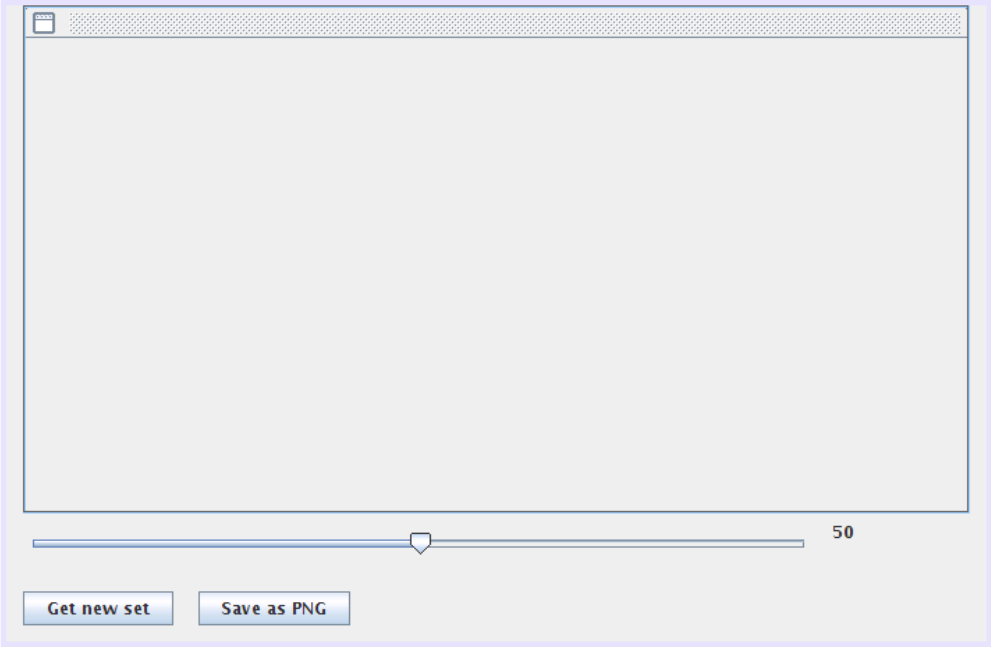

Fig. 23: Interfaz de usuario del programa visto desde el editor NetBeans

La primera versión del programa mostraba por pantalla la comparación entre una muestra que se denominaba la “modelo” y otra muestra del array de entrada a elegir mediante un *slider*. Este array se construía recogiendo MAX_MSG tandas de muestras recibidas y extrayendo los datos en formato adecuado para insertarlos en un *dataset* que, al estar ligado a un gráfico, se actualizaría automáticamente.

Para poder dibujar correctamente el gráfico, era necesario saber cuantos mensajes componían una sola tanda de muestras. Además, como se ha comentado anteriormente, se había elegido un tamaño total de 576 muestras, pero lo más adecuado era buscar una programación que se adaptara a cualquier número de muestras que se pudieran tomar, porque este número se varió a lo largo de las pruebas. Para ello, la solución tomada fue añadir al mensaje enviado vía radio desde el TelosB un flag que marcara cuando se trataba del último mensaje de la cadena, y simplemente leyendo este flag en el PC se podía saber que todas las muestras anteriores provenientes del mismo mote y que no tuvieran marcado este bit pertenecían a la misma cadena de muestras. Para esto, se hacía la suposición que los paquetes llegaban en el mismo orden en que se habían enviado. Esta hipótesis era válida puesto que sólo había un mote enviando los datos al PC, y los paquetes se enviaban uno a uno conforme se acababan de llenar de las 48 muestras necesarias.

Para poderlos mostrar correctamente, los datos leídos se han de convertir los datos a las unidades adecuadas, tanto temporales como de tensión. La separación entre muestras es igual al tiempo de muestreo, que aunque una opción para hacer que éste se actualice automáticamente si se cambia es enviar el utilizado en cada paquete, esto supone un *overhead* innecesario puesto que es un valor que no debe cambiar ni a lo largo del funcionamiento del nodo ni entre diferentes nodos, así que se opta por definirlo como una constante dentro del código. Para convertir la medida a tensión, se utilizaba el hecho que la tensión máxima son 3 V, de manera que se puede convertir de la siguiente forma:

$$Medida[V] = Medida \cdot \frac{3V}{2^n}$$

Donde n es el número de bits utilizado para cada muestra, 12 si se utilizan todos los que se pueden obtener del ADC.

Aprovechando otra de las funciones de JFreeChart, se añade posteriormente una función para poder salvar el gráfico en formato PNG. Tiene la ventaja sobre JPEG que no tiene pérdidas por compresión, de manera que los gráficos se guardan con mayor calidad.

El funcionamiento a nivel de usuario consiste simplemente en presionar “Get new set”, momento en el que el programa comenzará a recopilar datos. Cuando obtiene los suficientes, en el panel reservado de la pantalla, que se puede ver en la figura 23 sobre el *slider*, se refleja la última serie de muestras de los datos y el *slider* permite cambiar entre todas las series de muestras recogidas. Para salvar la en formato PNG solo es necesario pulsar el botón “Save as PNG” y creará un archivo con esta extensión en el directorio de ejecución.

El siguiente paso es visualizar las lecturas de ultrasonidos para comprobar el funcionamiento.

4.2.2 Verificación del funcionamiento y recepción de datos y programación del emisor

Para comprobar que se reciben datos tanto en el nodo como en el ordenador como fase preliminar se decide activar el modo lectura externamente con un mensaje de sincronismo enviado desde otro nodo y capturar ruido para verlo en el programa del PC. Una vez visto esto, es interesante intentar capturar una lectura correspondiente a algún emisor de ultrasonidos, por lo que se empieza a preparar el camino hacia el diseño de emisor y receptor por separado.

El montaje del circuito receptor en este momento consiste a grandes rasgos en un amplificador lineal que no llega a la zona de saturación y un filtro a la frecuencia de ultrasonidos que se utiliza, 40 KHz. Se tiene el propósito de aumentar el valor de salida de la señal del transductor antes de leerlo desde el TelosB, aumentando el alcance con el que se podía trabajar.

Al refrescar el gráfico capturado varias veces el programa se quedaba sin memoria, así que se analiza el comportamiento con el *profiler* de NetBeans. Cada vez que se actualiza este gráfico crea una nueva instancia, ocupando más y más memoria progresivamente. Para solucionarlo se varía el código para actualizar directamente el *dataset* vinculado sin crear nuevas instancias, y por las propiedades de JFreeChart esto ya altera el resultado de pantalla. El gráfico obtenido de la prueba de ruido se muestra en la figura 24, aunque en este caso la escala temporal no se encuentra aún ajustada, y solo sirve como ejemplo de primer gráfico obtenido vía las lecturas enviadas por el TelosB.

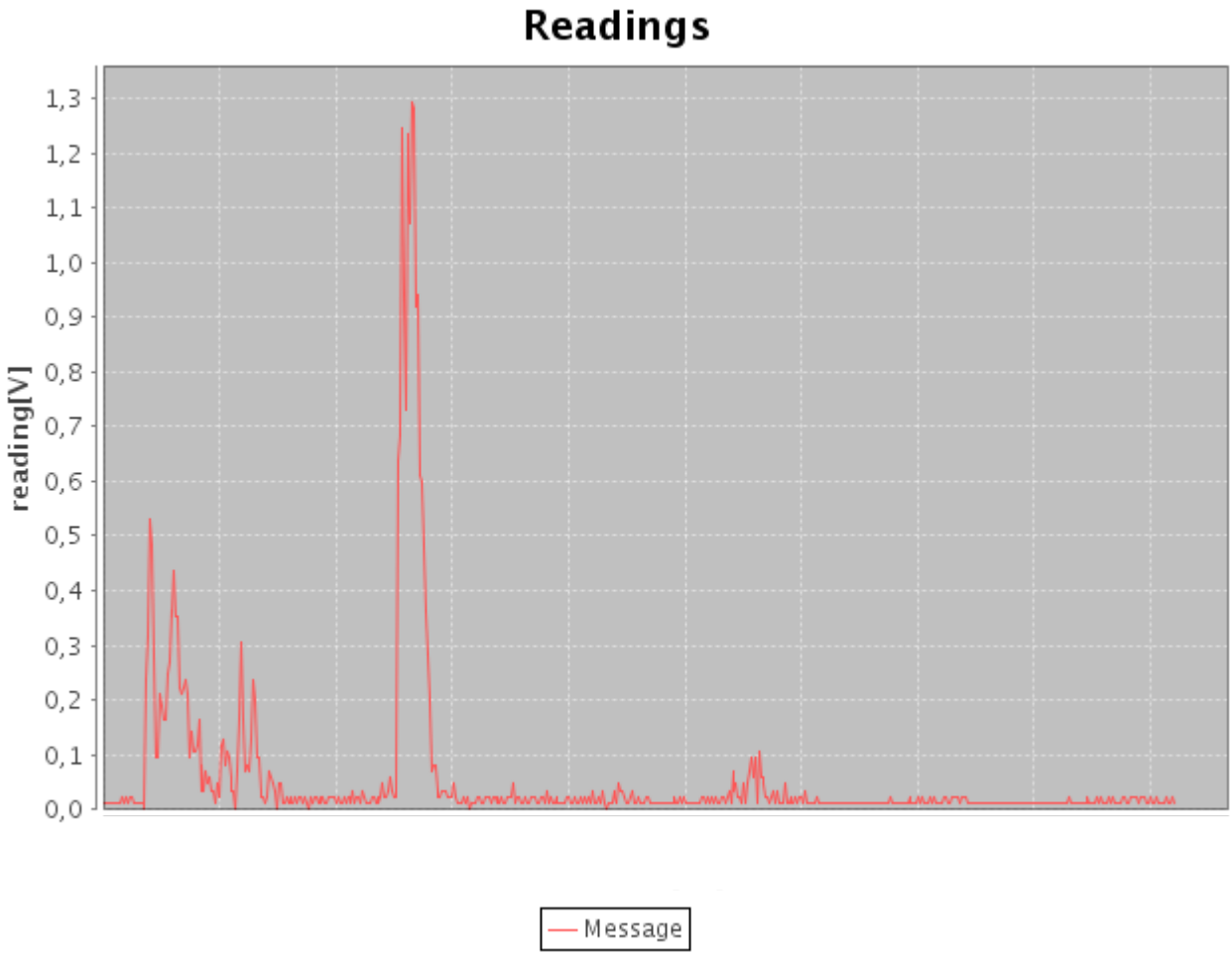

Fig. 24: Prueba de recepción de ultrasonidos con 12 bits

En este momento del proyecto de adaptación no se tiene ni la versión en TinyOS 2.x del código necesario para emplear el emisor de ultrasonidos, ni el circuito necesario para acoplar el emisor de ultrasonidos al mote, y por tanto se utiliza un TelosB con la programación y la implementación de otro proyecto, hecho en TinyOS 1.x. Lo que hace éste es enviar una señal de sincronismo por la radio y seguidamente enviar un pulso de ultrasonidos. Para aprovechar esto, la programación del *mote* del proyecto se modifica para esperar hasta recibir el pulso de sincronismo y entonces empezar a leer datos del conversor analógico-digital del TelosB. Para que la radio de uno y otro sean compatibles es necesario utilizar el mismo canal, añadiendo una opción de compilación en el proyecto con tal de utilizar el mismo que el usado en TinyOS 1.x por defecto.

Como se ha comentado antes, en primer lugar se realizan pruebas con el ADC trabajando con 12 bits. Esto da una resolución teórica de 0.73 mV, pero para captar esto hacen falta 16 bits por cada muestra, lo que es ineficiente puesto que hay 4 bits que no es están utilizando pero son ocupados. La segunda opción, comentada en el apartado 4.1.1 , es despreciar los 4 bits menos significativos antes de enviarlos por radio, de manera que cada muestra ocupe 8 bits. Esto permite guardar en un formato *uint8_t*, entero de 8 bits, pero reduce la resolución teórica a 11.72 mV. No obstante, considerando que el ruido en su mayor parte es menor que la resolución obtenida por utilizar 8 bits en lugar de 12, se decide utilizar esta última opción para reducir a la mitad el número de mensajes radio a enviar sin pérdidas importantes. Se ha de notar no obstante que esta reducción en el número de bits por muestra solo se dará mientras se envíen todas las muestras vía radio. Cuando se implemente el detector de picos, esto dejará de ser necesario puesto que los valores concretos de las medidas no interesan, si no únicamente el instante en que se detectan los ecos.

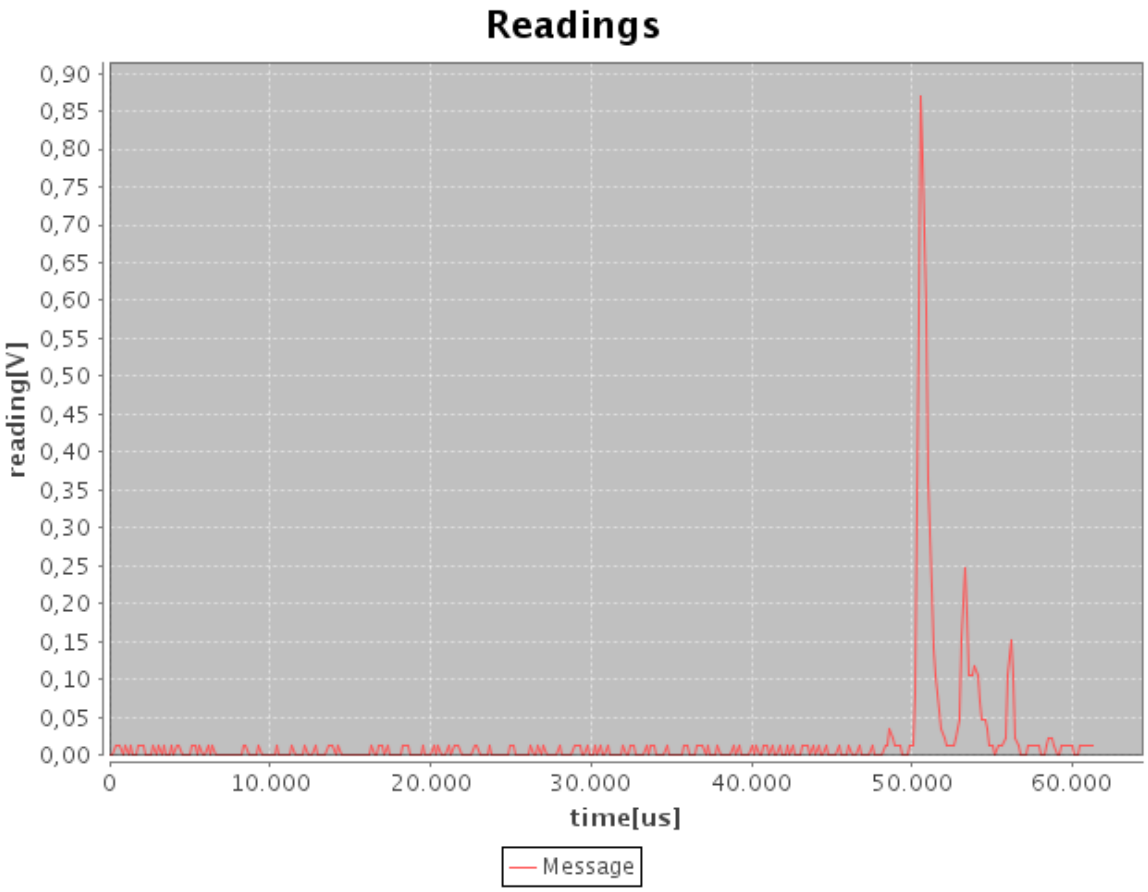

Fig. 25: Representación de la lectura de ultrasonidos de un pulso con varios rebotes con una resolución de 8 bits

Para realizar el gráfico de la figura 25, se tiene el emisor apuntando hacia un obstáculo paralelo a la mesa del laboratorio y el receptor en la mesa propiamente dicha. En esta representación el retardo inicial hasta la llegada del primer pulso no se ha de considerar como válido, puesto que la programación del emisor incluye un tiempo de espera variable superior a $40000\,\mu s$.

En este momento solo se descarta que el retardo sea provocado por errores en la programación del mote añadiendo *timestamps* a los mensajes enviados, comprobando que llegan en orden y que se ha comenzado a escribir cada mensaje con el retardo adecuado, así como comprobando el retardo en comenzar a leer muestras desde que se ordena empezar el muestreo. Para esto se añade código para imprimir los retardos mencionados por pantalla. En todos los casos los intervalos de tiempo encontrados coinciden casi perfectamente con los intervalos esperados, y eso teniendo en cuenta que el hecho de añadir instrucciones para mostrar retardos o guardarlos en ocasiones interfiere en el tiempo necesario para realizar otras acciones.

Descartado el receptor, sólo quedaba que el retardo fuera causado por el emisor. En la programación del emisor de otro proyecto utilizado en este momento, éste realmente trabajaba como emisor/receptor, y necesitaba de un tiempo para hacer el cambio entre los dos modos de funcionamiento. Se intenta modificar la implementación de este emisor, pero cambiar el retardo provoca un error interno debido a la falta de tiempo para hacer el cambio de modo, por lo que se opta por dar los resultados como válidos en espera de poder utilizar el emisor propio programado íntegramente en TinyOS 2.x. Este retardo se corrige y explica con detalle más adelante en el apartado 5.1.2 .

Cuando se comprueba que tanto la comunicación PC-mote es funcional como que los resultados obtenidos por la lectura de datos son aceptables, el siguiente paso es programar la detección de picos.

5 Implementación de la solución

En los apartados anteriores se ha visto en primer lugar la tecnología que se va a utilizar, consistente en un nodo Crossbow TelosB que se programa en TinyOS para interactuar con los diferentes emisores y receptores utilizados en la duración del proyecto. En segundo lugar, se ha visto como se implementaba el primer nodo con un programa que recoge las muestras tomadas por el conversor analógico-digital del TelosB y las envía vía radio para ser recogidas en el PC, donde se mostraban por pantalla. Lo siguiente es implementar la solución propiamente dicha para los dos casos distintos que se han de tratar: cuando se pretende localizar blancos que podrán portar un nodo y por tanto el emisor y el receptor serán nodos distintos, y cuando se pretende localizar blancos que no podrán llevar ningún identificador, y por tanto emisor y receptor serán el mismo nodo.

Para cada caso se hablará de la implementación y de los resultados obtenidos.

5.1 Primer caso. Blancos con identificador. Entornos abiertos.

Tener emisor y receptor en nodos separados es importante para el caso en que el blanco pueda portar algún tipo de dispositivo emisor o receptor. Al tener un elemento localizador es más fácil discriminar entre blancos válidos y ruido.

El funcionamiento consiste en enviar un paquete de sincronismo seguido de un pulso de ultrasonidos, de manera que el receptor conoce quién le esta enviando la información y la distancia hasta el nodo emisor, con la ayuda de los ultrasonidos. Para ello se deberá programar el algoritmo de detección de picos del que se habla en el apartado 2.5 , que ya será válido para el caso en que emisor y receptor se encuentren en un mismo nodo.

Un escenario en que puede darse esto es, por ejemplo, un grupo de ciclistas, donde las bicicletas pueden llevar emisor y receptor para localizarse entre ellas.

5.1.1 Programación del receptor. Implementación del detector de picos.

Como se comenta en el apartado anterior, se decide crear una solución específica para el proyecto, utilizando un algoritmo que detecte los blancos en los que rebota la señal cuando emisor y receptor están en el mismo nodo o la distancia desde la que se envía el pulso de ultrasonidos cuando emisor y receptor están en nodos distintos y separados. El algoritmo elegido para su implementación en ese mismo apartado consiste en el circuito digital de la figura 26.

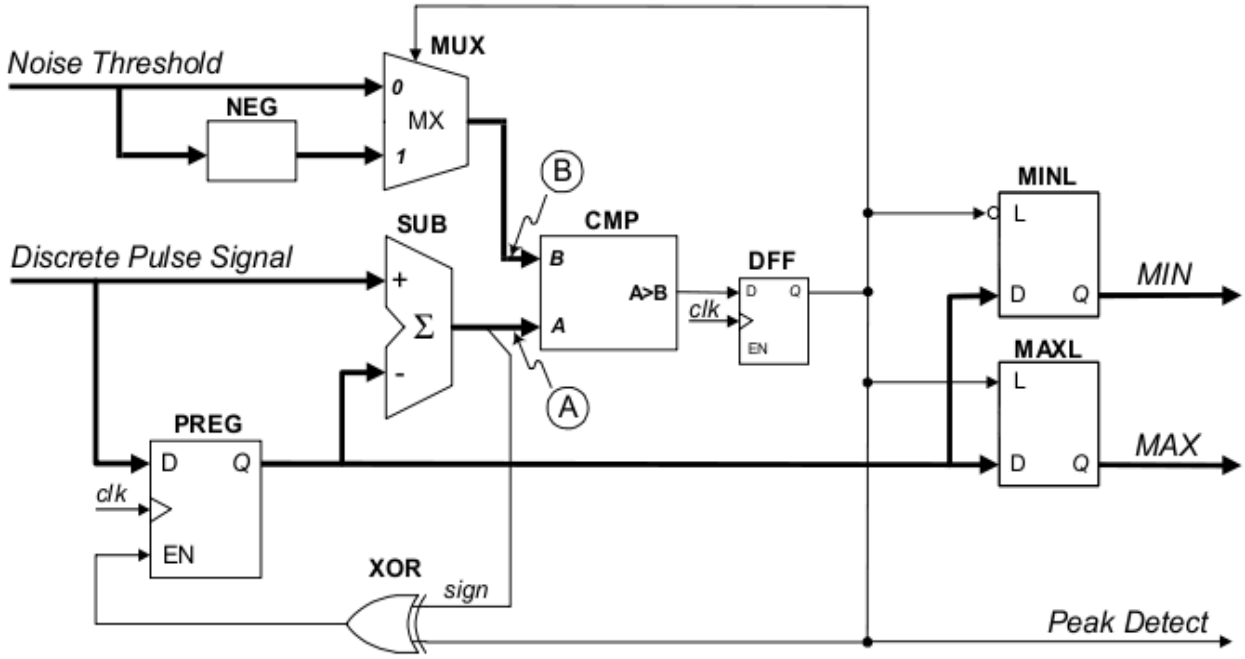

Fig. 26: Esquema del detector de picos utilizado.
(fuente: [17])

Los bloques son todos de sencilla traducción individualmente, exceptuando el registro PREG que requiere una función específica. Este registro, cuando está activado (*enable* = 1) compara la entrada con el registro interno y asigna tanto a la salida como al valor de la memoria interna el mayor de los dos. En el caso que esté desactivado, el comportamiento es similar solo que se pone en el *output* y se guarda el valor menor de los dos.

No se implementa la parte de guardar el valor máximo y mínimo simplemente porque es una ocupación de memoria innecesaria, ya que lo único interesante es el instante de detección. Por esto mismo ya se prepara el sistema para no enviar todos los datos muestreados, puesto que se ha visto que los datos se leen por el ADC de manera correcta, si no que únicamente se enviarán por radio los valores de los instantes donde se han encontrado los picos. Además así se evita tener que enviar tantos mensajes radio y de un tamaño tan elevado, puesto que es mucho más fácil compactar la información constatando únicamente si se en un instante dado hay un máximo o no.

La primera versión consistía en enviar un valor con la posición en número de muestra del pico. Esto es fácil de convertir a tiempo puesto que sólo es necesario multiplicar esta posición con el tiempo de muestreo. Esto tenía el problema que se limitaba la cantidad de picos que se podía enviar de manera arbitraria, y no siempre sería posible mandarlos todos en caso que hubiera demasiados.

Para evitar esto y reducir el tamaño del paquete, lo que se hace es enviar un bit por cada muestra de las tomadas, de manera que cambiar el número de muestras cambia automáticamente el tamaño de este campo. Así se tenía la ventaja que es posible contar cualquier número de picos encontrados, sin limitaciones arbitrarias. Para facilitar la interacción con los datos se añaden macros de procesado de bits tanto en el código del nodo como en el PC, y se decide que un uno en un bit señala la posición de un pico en la muestra correspondiente.

0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Fig. 27: Ejemplo de paquete radio

Por ejemplo, suponiendo que hubiera 32 muestras, podría resultar un paquete como el de la figura 27, de 32 bits. En este caso, y teniendo en cuenta que en programación la primera posición de un *array* es 0, se habrían detectado únicamente dos picos en las posiciones 10 y 22, lo que querría decir que los instantes correspondientes a las muestras 10 y 22 tienen un pico. Para encontrar el instante del pico bastaría con multiplicar el número de muestra por el tiempo de muestreo.

Una vez se tiene el receptor es hora de programar el emisor para utilizar el transductor propio.

5.1.2 Programación del emisor. Implementación de la comunicación radio.

Para diseñar el software del TelosB que se encarga de manipular el emisor de ultrasonidos, con tal de adaptar el funcionamiento del *mote* tanto al circuito del transductor emisor como al receptor previamente programado, lo que se hace es emitir un paquete de sincronización seguido del cambio de uno de los pines a nivel alto durante el tiempo que se deseara que durase el pulso. Una vez hecho esto, debía comprobarse que los resultados eran los esperados. Nótese que el circuito que controla al emisor tiene una entrada que provoca la emisión de un pulso modulado a 40 KHz mientras esta entrada esté a nivel alto (3V).

El primer paso para la implementación es la inclusión del sistema de radio, que emite el paquete de sincronización a intervalos regulares programables. Tras esta emisión se pone un pin de salida a 3V durante un intervalo de tiempo arbitrario, adaptable según las necesidades en cada momento del proyecto. El sistema de radio que se utiliza es básicamente el mismo que para el caso del receptor, y para la programación del pin de salida se usa un interfaz en TinyOS que trabaja directamente con el puerto de entrada/salida del controlador específico de la placa del TelosB. Para elegir el pin adecuado se debe tener en cuenta que no todos tienen capacidad de trabajar como *outputs*, y que en ocasiones se tiene que llamar por el número de puerto (p ej. Port62) en vez de por su nombre (en el mismo ejemplo, ADC2). Una vez se elige el pin, sólo queda configurarlo como salida y provocar que esté activado durante un tiempo tal que sea lo suficientemente ancho como para ser captado adecuadamente por el muestreado configurado a $160\mu s$ que se ha comentado anteriormente pero no tan ancho como para reducir en exceso la precisión de la medida. Tras varias pruebas se elige este tiempo como $250\mu s$. Para ver que efectivamente se produce un pulso del tiempo programado se conecta la salida al osciloscopio, donde puede comprobarse que el comportamiento hasta ahora es el esperado.

Para poder mirar con más detalle y sin ningún tipo de interferencia externa los retardos se realizan las pruebas sin los transductores de ultrasonidos ni los circuitos, conectando simplemente el pin de salida del nodo emisor al pin de entrada del nodo receptor, que se ocuparía de las lecturas y detección de picos. Así, el envío del pulso será, a todos los efectos, instantáneo, con lo que se pueden medir los retardos que actúan sobre los TelosB sin el efecto de los circuitos intermedios.

La comprobación consiste en realizar la conexión emisor/receptor antes mencionada con un cable y cargar el programa mencionado en los apartados anteriores que envía todas las muestras al PC para ver el gráfico. Al realizar esto no se obtiene ninguna lectura en el gráfico, por lo que se ha de discernir qué tipo de retardo está afectando al programa. Para investigar se alarga el pulso emitido hasta unos 10 ms para descartar que no se estuviera leyendo de ninguna manera en el receptor. No obstante, el final del pulso de ultrasonidos no se capta siempre en el mismo instante, sino con unas variaciones de hasta 8 ms en el instante de finalización, lo que indica que hay algún tipo de retardo aleatorio que afecta de manera global. Esto implicaría equivocarse en la posición del emisor con respecto al receptor en hasta 2.77 m. Como esta variación no es aceptable, se pasa a investigar las posibles causas. A continuación se detalla el procedimiento para descartar las diferentes posibilidades.

Al estar conectados pin a pin, no hay posibilidad que sea por los circuitos del emisor o del receptor, así que hay que descartar otras posibilidades para poder solucionar el error. Se pasa por tanto a comprobar si el retardo aleatorio se produce por limitación hardware a la hora de activar el pin, por el código programado o por el código interno de TinyOS.

Para lo primero, si bien es posible que el pin no se encienda inmediatamente tras dar la orden por causas de hardware, es altamente improbable que esto provocara un retardo aleatorio, así que se pasa a estudiar el código.

En el caso del código implementado, primero se descarta que solamente se viera afectado el tiempo de apagar el pin, para lo que se añade un retardo controlado para poder ver por pantalla como empezaba y finalizaba el pulso. Con esto se ve que el pulso mantiene en todo momento la duración, y por tanto el único problema es el instante de comienzo de captación. La manera de ver que el emisor no es el causante del error es utilizando un Counter con una precisión igual al reloj interno de 32 KHz. Se elige este componente en concreto porque se guarda el instante de tiempo en que se está produciendo algo de forma asíncrona, de manera que se ejecuta en el mismo instante en que se llama, sin entrar en la cola de *tasks*. Para interferir al mínimo en la ejecución del código se añaden unas variables que almacenan los instantes de tiempo que se consideran críticos. Estos valores son mostrados por pantalla más tarde, una vez ha terminado el ciclo de envío de paquete de sincronismo y pulso de ultrasonidos. En concreto, se comprueba el tiempo que tarda en enviarse un mensaje radio, así como el tiempo desde este instante hasta que se pone la salida a 3V. Tras una serie de pruebas, los tiempos en ambos casos resultan ser constantes, con lo que se descartan.

Se contempla la posibilidad que sea el receptor, así que se realizan nuevas pruebas de temporización, mostradas a continuación:

```
Timings
 timeSyncReceived = 52783, dt = 9
 timeCalledGetData = 52792, dt = 80
 timeStartedGetData = 52872
Timings
 timeSyncReceived = 49855, dt = 8
 timeCalledGetData = 49863, dt = 81
 timeStartedGetData = 49944
Timings
 timeSyncReceived = 46653, dt = 9
 timeCalledGetData = 46662, dt = 81
 timeStartedGetData = 46743
Timings
 timeSyncReceived = 43599, dt = 9
 timeCalledGetData = 43608, dt = 80
 timeStartedGetData = 43688
```

Para cada línea se muestra el tiempo en ciclos de reloj en tres puntos clave del código, a saber: *timeSyncReceived* corresponde al tiempo en que se lanza el evento correspondiente a un paquete de sincronismo recibido, *timeCalledGetData* corresponde al momento en que se pide al ADC que empiece a muestrear y *timeStartedGetData* corresponde al instante en que el ADC finalmente toma el primer grupo de 16 muestras. Cada *Timings* indica una nueva tanda de muestras, y *dt* indica el número de ciclos de reloj entre la línea mostrada y la siguiente.

Como puede verse, desde que se recibe el pulso de sincronización hasta que empieza realmente a leer datos pasan únicamente entre 8 y 9 ciclos de reloj, lo que supone entre 0.25 ms y 0.28125 ms. Los 80 u 81 ciclos de reloj pasados desde que se pide hasta que empieza el sampleo corresponden al tiempo de realizar 16 muestras, es decir, 2.5 ms.

Así pues, solo queda observar el código utilizado por la base de TinyOS. Se miran los componentes

tanto del Msp430 como del CC2420, que son el microprocesador y la radio del TelosB, respectivamente. Al mirar el funcionamiento de la radio se detecta que utiliza por defecto CSMA con un retardo aleatorio que coincide con el comportamiento del retardo que se ve por pantalla. Para determinar si efectivamente es la causa del problema se anula este mecanismo, lo que produce que los gráficos se vean desplazados en el tiempo pero por un valor constante, confirmando que el retardo aleatorio es provocado por este sistema. No obstante, anular el tiempo de *backoff* de CSMA no es la solución adecuada, puesto que aumentaría la probabilidad de colisión entre los paquetes radio cuando aumentara el número de *motes* en el sistema. Así, se pasa a utilizar el mecanismo de *backoff* de la radio para asignar el tiempo desde la capa de aplicación, pese a que esto resta transparencia al funcionamiento del sistema. Para interferir lo menos posible en el diseño de TinyOS, se utiliza la misma fórmula para calcular el tiempo de *backoff* que se encuentra en el kernel del sistema operativo, solo que se guarda el valor utilizado para añadirlo al tiempo de espera antes de activar el emisor de ultrasonidos. Con este cambio el resultado es el mismo que anulando por completo el tiempo de acceso al medio de CSMA, pero manteniendo la fiabilidad contra las colisiones radio.

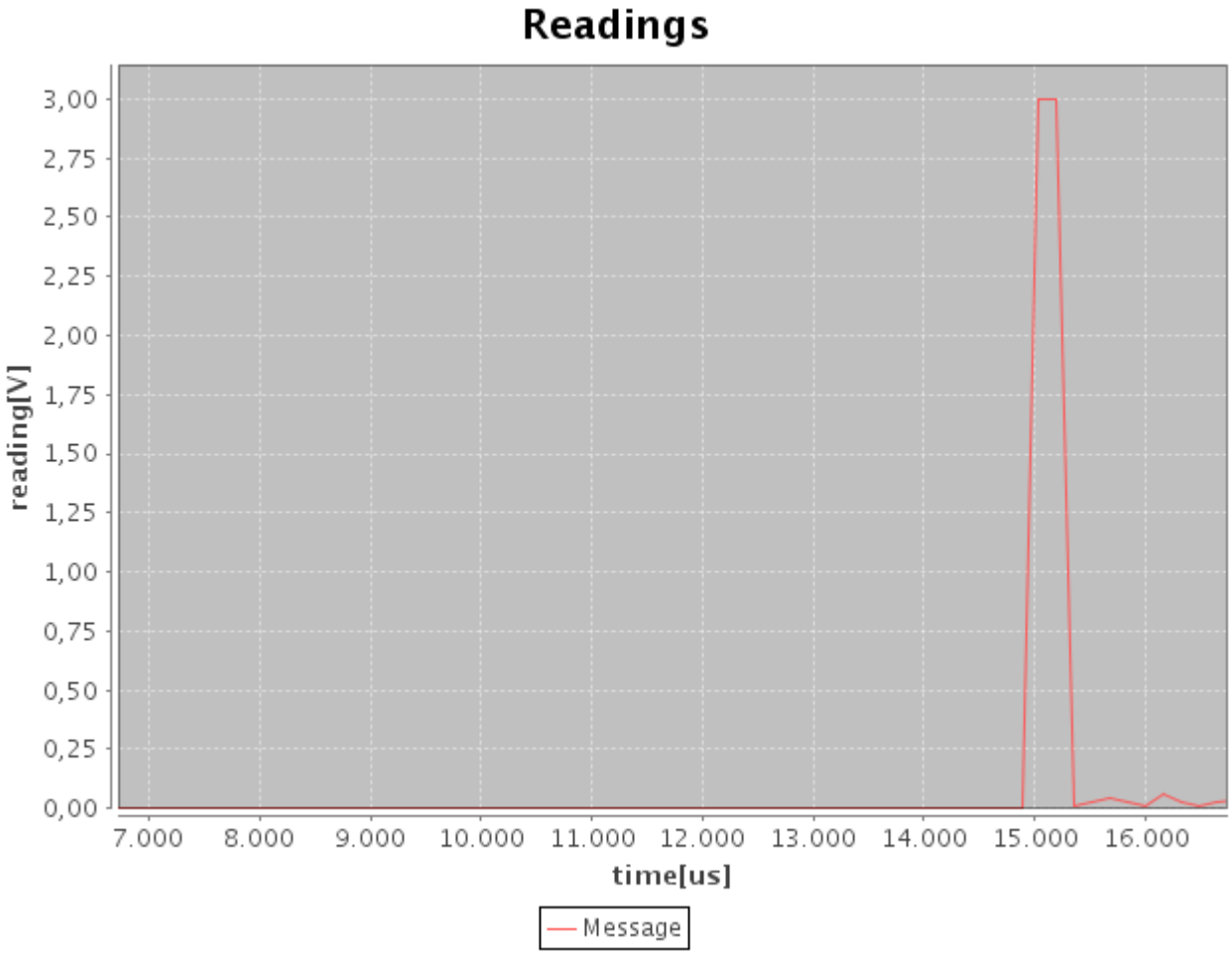

Fig. 28: Pulso de 250µs emitido con un retardo con respecto al paquete de sincronización de 18 ms.

Para valorar el efecto del retardo constante que afecta al sistema se hace que el pin de output se ponga a nivel alto 18 ms después de enviar el mensaje radio, momento en que se captura el gráfico de la figura 28.

Como se puede ver, se detectado que el pulso llega a 15 ms en lugar de 18 ms, lo que dice que el retardo constante es de unos 3 ms. No obstante, es posible añadir un tiempo de espera antes de enviar el pulso para que la primera muestra en la lectura corresponda a dos TelosB en el mismo sitio

exacto y por tanto compensar este tiempo.

Posteriormente, se pasa a ver el funcionamiento del algoritmo detector de picos realizado en apartados anteriores. Para ello, simplemente se añade el código necesario para imprimir por pantalla los resultados de la detección, con lo que se obtiene lo siguiente:

```
Peak detected: 4095, 0
Peaks detected:
peak 0 = 96, time = 15360 us
```

En primer lugar se enseña el valor del ADC en la muestra inmediatamente anterior a la decisión y en la muestra de la decisión en sí. Después se muestra en una lista el número de la muestra que ha lanzado esa decisión, así como el tiempo que corresponde a esa muestra. Como se puede ver, hay aproximadamente 3 ms de menos en el cálculo de tiempo, igual que en el caso del gráfico, con lo que indica que la detección de picos es coherente con el resultado esperado en la entrada del receptor.

Esto implica que, desde que se envía el paquete hasta ese momento, hay un retardo de cerca de 3 ms entre que se envía realmente el pulso y se determina su recepción. En todo caso, este tiempo se mantiene constante en todas las pruebas y es compensable vía software, así que no supone un grave problema.

5.2 Segundo caso. Blancos sin identificador. Entornos cerrados.

Una vez se tiene el sistema funcionando con emisor y receptor por separado se pasa al segundo caso comentado en la introducción, cuando el blanco no portará ningún elemento, y por tanto emisor y receptor han de ser coordinados por el mismo *mote*. Como se ha dicho antes, aquí se utiliza a modo de sonar. Pese a que emisor y receptor son dos transductores diferentes, se encuentran en un lugar lo suficientemente cercano para poder decir que a efectos prácticos es el mismo caso que si fueran el mismo transductor, con la ventaja añadida que como cada uno se encarga de un modo (emisión o recepción), no hay una distancia mínima teórica de captura de datos.

De esta manera, puede ser utilizado, por ejemplo, para realizar medidas de ocupación en una sala si se tiene un número suficiente de pares emisor-receptor y estos están sincronizados con un PC que recoja todos los datos y los compile. Por ello, se programan de manera que sea posible la comunicación o bien entre ellos o bien con un nodo que se ocupe de organizar el sistema.

Para este caso, el funcionamiento a programar es emitir el pulso de ultrasonidos utilizando un pin de salida y recibir los datos y muestrearlos utilizando un pin de entrada del mismo nodo. También se aprovecha que se tiene información sobre la temporización del nodo para realizar promediado con las medidas.

5.2.1 Cambios al tener emisor y receptor en el mismo nodo

Al unir el código de emisor y receptor en el mismo nodo ya no es necesario tener en cuenta el acceso por CSMA de la radio de TinyOS, de manera que se elimina esa parte para ahorrar memoria.

Como en TinyOS no existe la concurrencia, es imposible activar a la vez el pin de emisión o DAC y el de escucha o ADC, así que se opta por activar en primer lugar la recepción y en segundo lugar, la emisión del pulso. Se realiza en este orden debido a que si se tienen muestras de lo que ha sucedido antes de enviar el pulso, con no tomarlas en cuenta es suficiente, pero no puede repararse la falta de información que supone realizarlo en orden inverso.

Aunque para las pruebas de funcionamiento se utiliza solamente un nodo que cada 5 segundos toma una medida o serie de muestras y la envía vía radio al PC, también se deseaba poder pedir medidas

con un nodo organizador. Para ello se incluye la programación necesaria para realizar las medidas únicamente cuando llegue un paquete de sincronismo con su nodo como nodo destino, para evitar que varios nodos realicen las medidas a la vez y puedan molestarse con pulsos cruzados de ultrasonidos.

En este momento ya se cuenta con un emisor activado mediante el DAC del TelosB, así como un receptor[1] que posee un módulo de detección de pulsos vía hardware, de manera que la salida está a nivel bajo si no considera que hay un rebote y a nivel alto, saturando, si considera que sí lo hay.

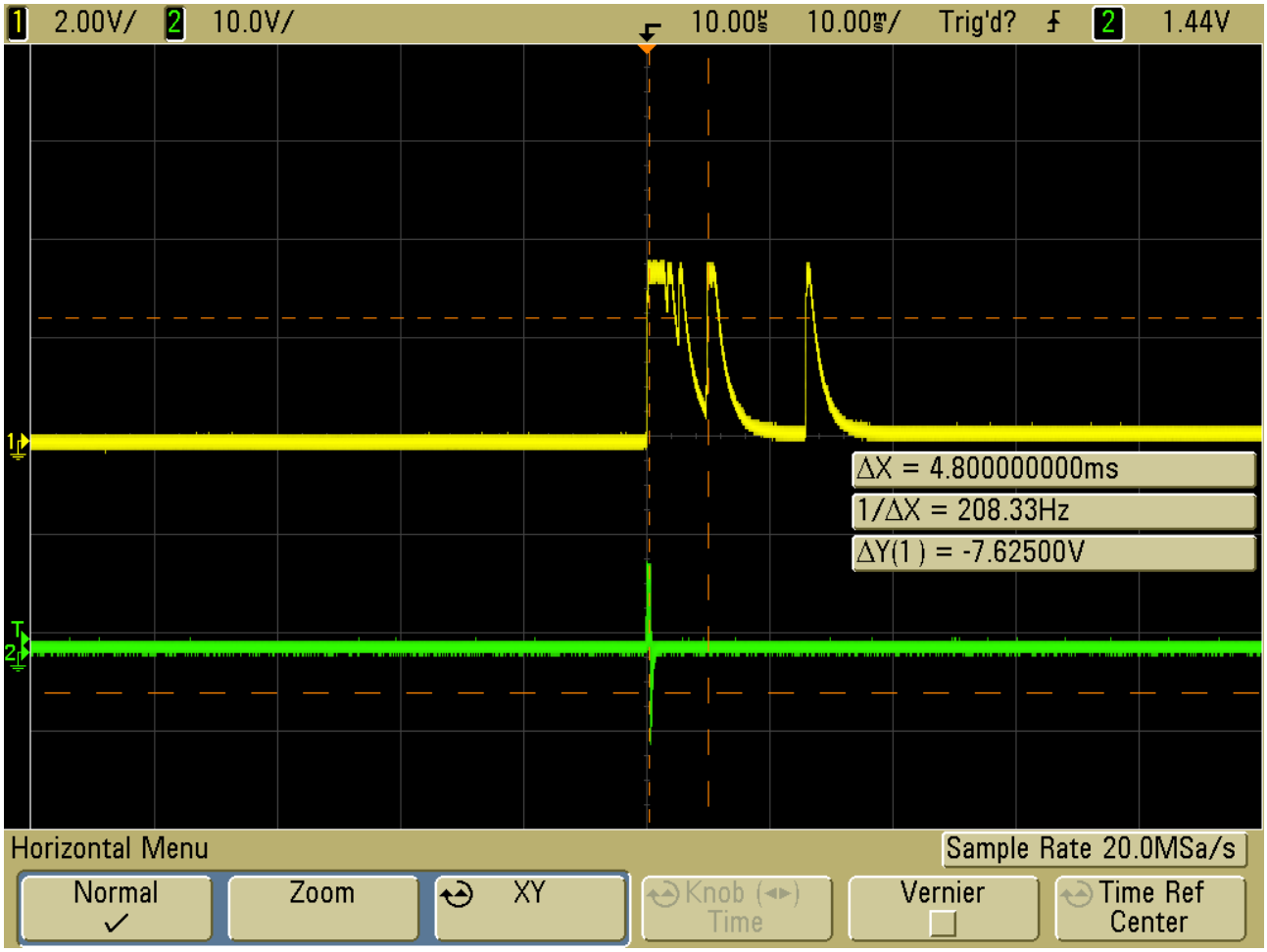

Fig. 29: Emisor (en verde) y receptor (en amarillo) capturados en el osciloscopio con dos blancos.

Como puede verse en la figura 29, al estar tan cerca se acopla el emisor al receptor, así que se ignoran los picos cercanos en el PC, puesto que provendrán de este factor.

Aunque este circuito receptor puede funcionar con un detector de picos software más sencillo, el implementado demuestra obtener los resultados correctos, adaptando los parámetros de umbral de ruido a valores muy elevados para compensar que el ruido se podía producir por detecciones falsas. Un umbral de ruido bajo no tiene sentido cuando se pasa de cero a saturación cada vez que se encuentra un pico, pero sí puede provocar problemas.

Como funciona correctamente, se mantiene este algoritmo para poder trabajar también con otro tipo

1 El receptor utilizado para estas pruebas es mostrado en un artículo en preparación. Este artículo habla de la implementación (particularizada para Vodafone) de un sistema de conteo de personas y cómo un sistema central recopila y estudia las medidas tomadas para posicionar a personas en la sala. Al realizarlo se utiliza un circuito que posee detección de picos por hardware, con lo que sólo tiene dos modos de funcionamiento, saturación y no saturación. Es interesante debido a que es capaz de detectar potencias de señal mucho más reducidas, con la contrapartida que el ruido es mucho más elevado.

de receptor que no sature a la salida, haciendo que la programación sea fácilmente adaptable a todos los casos.

5.2.2 Promediado de los resultados y modelo del escenario

Siguiendo en el caso de tener elementos que compartan emisor-receptor para detectar personas, se puede añadir una cierta habilidad al sistema para discriminar de manera más fiable ecos válidos de ecos inválidos. En este caso, por ejemplo, es interesante que los elementos estáticos de la habitación no sean contados como rebotes válidos, y que los ecos espurios provocados por ruido o similar no se tengan en cuenta a la hora de calcular ocupación. Para ello, se propone encontrar el modelo de la habitación, y promediar los resultados obtenidos tras varias repeticiones.

Encontrar el modelo consiste en observar y guardar los picos generados por el mobiliario y otros objetos, con la sala a estudiar vacía, para posteriormente eliminar estos picos en las medidas de ocupación. Aunque esto es posible hacerlo desde el PC, eliminarlos en cada nodo distribuye el procesado a realizar agilizando la compilación de resultados. Como se puede dar la posibilidad que haya algún pequeño *jitter* en la obtención de las medidas, también se tienen como picos las muestras inmediatamente adyacentes al pico real.

Así, se busca realizar algún tipo de promediado en las muestras o en los picos para descartar los ocasionados por el ruido. Como contrapartida, es necesario destacar que promediar hace que no se detecten los blancos en movimiento, pero para el caso de este proyecto es preferible poder situar el número correcto de personas en el lugar adecuado.

En este caso, se usa un algoritmo de promediado sencillo, consistente en realizar la media de varias series de medidas. Ocupar el mínimo de memoria posible es un objetivo muy importante, puesto que el *hardware*, como se ha visto, está limitado en este aspecto. Por ello cada vez que se captura una secuencia entera de muestras, y antes de pasar a recibir la siguiente, se divide entre el total de veces que se va a realizar la media y se suma a un vector que mantiene el resultado acumulado. De esta forma, una vez se ha realizado el número de medidas correspondiente se puede utilizar este vector como entrada de la función de detección de picos, liberando memoria. Utilizar una solución más convencional guardando todas las muestras antes de hacer la media es un coste de memoria innecesario.

En ocasiones alguna de las series de muestras aparece afectada por un desplazamiento temporal con respecto a las otras series. Si no se elimina este efecto, como el desplazamiento sólo afecta a una de las repeticiones, el promediado puede provocar la eliminación de picos válidos al no encontrarlos en todas las series. Para evitar esto se añade un contador que mida el tiempo pasado entre que se activa el pulso y realmente se efectúa la activación. Cuando este tiempo es distinto de un valor configurado de antemano, esa cadena de muestras no se tiene en cuenta y se repite. Esto no es factible en el caso que el emisor y el receptor estén por separado puesto que requeriría enviar mensajes cada vez que alguna muestra saliera mal, aumentando la ocupación del canal. Además, en un escenario donde los blancos serán móviles promediar no ayuda a situar la posición.

Para evitar utilizar la radio de manera innecesaria, se aprovecha de la manera de codificar los picos con bits hallada en el apartado 5.1.1 para guardar una medida como modelo del canal. Así, cuando un nodo se enciende o cuando recibe un paquete de sincronismo con un *flag* que determine que debe iniciar la adquisición del modelo del canal este nodo hace una serie de medidas, detecta los picos y los guarda en un espacio de memoria aparte. Entonces cada vez que se pide una medida normal, se comparan con ella pico a pico con una máscara de bits, eliminando los que coincidan. En el caso de que no quedara ningún pico después de la comparación, no se envía el paquete radio para ahorrar baterías.

No obstante, esto puede llevar a equívocos, puesto que puede darse que un nodo lleve mucho

tiempo sin contestar debido a la falta de baterías, un fallo en el canal de comunicación o una avería, y no debido a que no encuentre medidas nuevas. Por ello se añade un temporizador que envía un paquete de sincronización si detecta que lleva más de un determinado tiempo (configurable) sin enviar un paquete de picos.

5.3 Uso del emisor comercial SRF02

Aún en el caso de emisor y receptor controlados desde el mismo nodo se decide emplear como emisor el hardware comercial SRF02.

Para esto se cambia el código de manera que se pueda alternar entre utilizar el SRF02 o el circuito emisor propio. Además se aprovecha para hacer pruebas con otros dos tipos de circuitos emisores distintos: uno que únicamente filtra, amplifica y detecta la envolvente de la señal y otro que amplifica hasta saturar después de filtrar.

5.3.1 Conexión del SRF02 al nodo

Una vez se tienen los drivers del interfaz I2C y, por tanto, compatibilidad con la radio, se pasa a añadir modificaciones al código para poder utilizar indistintamente tanto el emisor activado por el DAC como el emisor del SRF02 en modo de sólo enviar pulso. Para ello, se añade una opción de compilación denominada USE_SRF02 para poder cambiar entre ambos modos de funcionamiento.

Cuando esta opción está definida, todo la parte del código innecesaria para el funcionamiento con el SRF02 queda comentaba, y viceversa, asegurando que se usan los recursos estrictamente necesarios para cada caso. Se decide separar el código en vez de eliminarlo completamente para no perder la capacidad de utilizar algún tipo de emisor activado por el DAC.

Los drivers del SRF02 programados se ocupan de los recursos necesarios para poder enviar información a través del puerto I2C. Concretamente, como parte de su funcionamiento, se ocupan de activar y desactivar el sistema de radio según se necesita, por lo que se pide realizar una medida antes de empezar y así inicializarla. De todas las funciones del SRF02 se elige una llamada para conseguir el valor de su firmware dado que es una medida que no requiere ni de la emisión de un pulso de ultrasonidos ni del uso de la memoria del TelosB, puesto que no es necesario guardar esta información.

A la hora de realizar la media de las medidas, comentada anteriormente, es necesario tener en cuenta que el dispositivo necesita aproximadamente 66 ms entre peticiones, según las especificaciones de la documentación. Por ello se modifican los drivers para que en lugar de 500 ms entre medidas, como estaba especificado, permita hacerlas tan rápido como sea, pero por seguridad se añade un temporizador en el programa principal para realizar las medidas cada 100 ms.

5.3.2 Comprobación de resultados

Una vez se tiene el SRF02 funcionando se realizan nuevas pruebas de medidas con los dos circuitos receptores mencionados anteriormente. En primer lugar, se conecta al TelosB el que únicamente filtra, amplifica y detecta la envolvente.

Pese a la amplificación, la salida es demasiado baja, no teniendo un rango total superior a los 40 cm ida y vuelta. Se toma la captura de la figura 30 poniendo dos obstáculos frente al emisor a corta distancia.

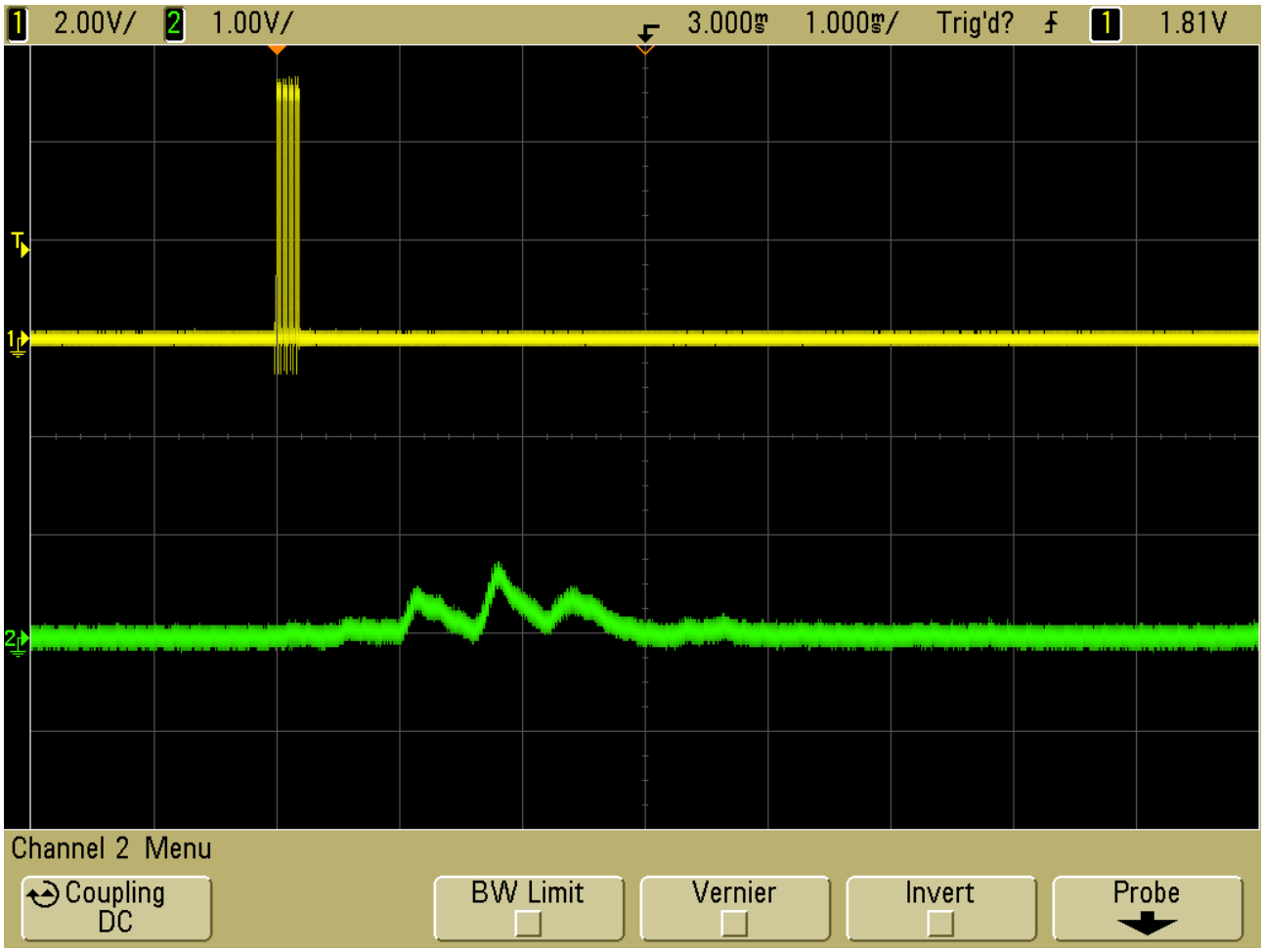

Fig. 30: Captura del osciloscopio después del detector de envolvente

Al hacer esta captura, el TelosB envía al PC la siguiente información:

```
Peaks detected, node 7
0000 0000 1210 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 - Peak #0 = 5760.0 us, distance = 1.97568 m.
 - Peak #1 = 6560.0 us, distance = 2.25008 m.
    - Peak #2 = 7040.0 us, distance = 2.41472 m.
```

El primer renglón de números indica en qué posición se encuentran los picos, donde cada cifra representa 16 bits en hexadecimal. Esto es según la codificación utilizada en el mensaje para guardarlos. Para más información referirse al anexo 8.3.3 . Lo siguiente que se presenta son los picos encontrados, determinados tanto en tiempo como en la distancia equivalente (ida y vuelta). Como se puede ver, hay un desfase temporal entre el gráfico y los picos detectados que implica un desfase en la posición. Comparando los tiempos del primer pico este desfase se calcula como unos 4.4 ms, y es debido principalmente al tiempo que se tarda en comunicar con el SRF02 y el tiempo que tarda éste en activar el pulso. No obstante, la separación entre ellos (800 µs y 480 µs), es decir, el tiempo relativo, coincide con el obtenido experimentalmente en el osciloscopio.

A continuación se prueba con el segundo circuito receptor, al cual para intentar obtener un mayor alcance, se añade otra etapa de amplificación. Esto provoca la saturación a la salida del circuito, así como mayor sensibilidad.

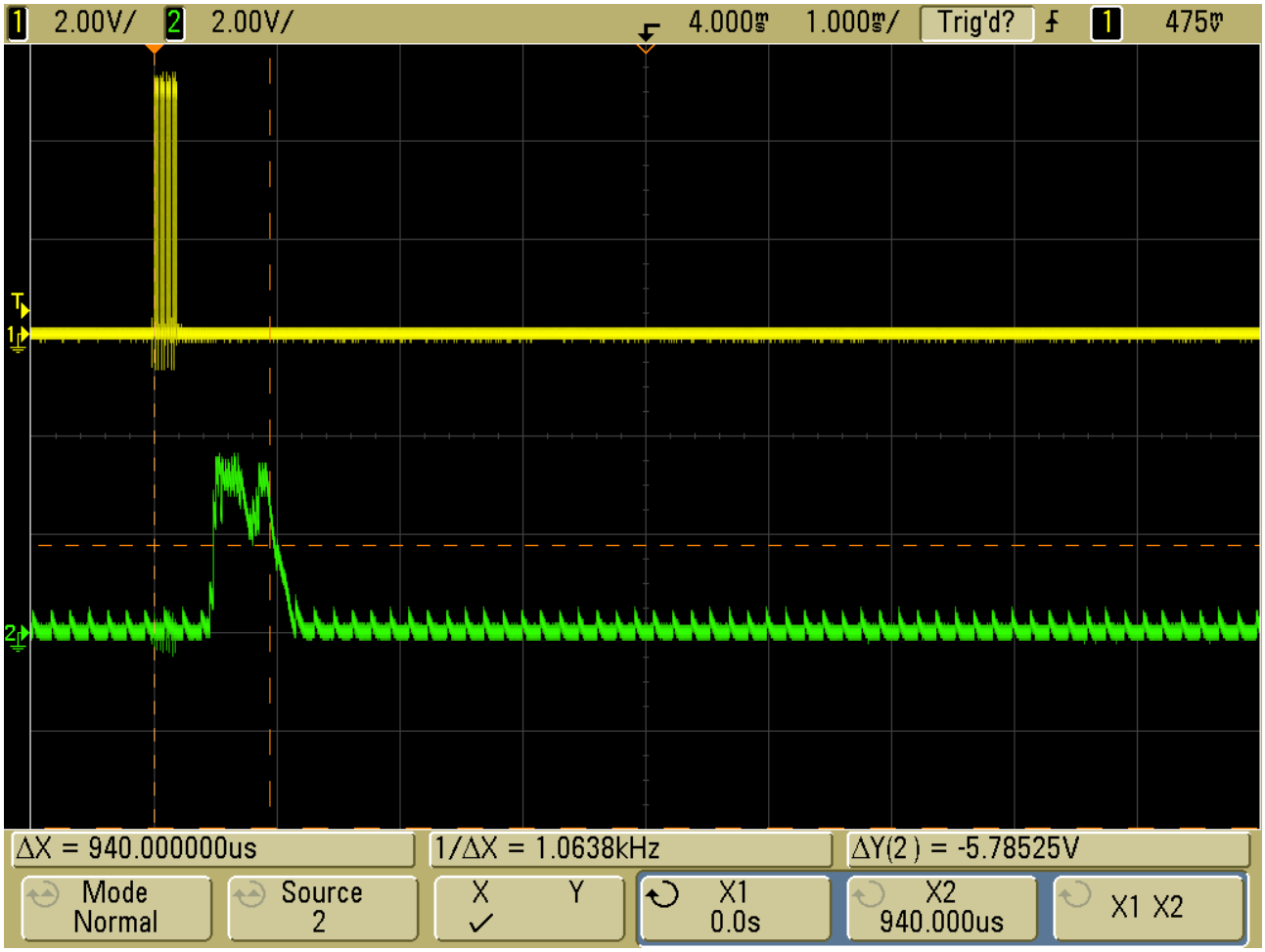

Fig. 31: Acople entre el emisor y el receptor, captura al aire

En la figura 31 se puede ver como la simple cercanía entre emisor y receptor ya causa un a recepción a unos 940 µs. Como el modelo del canal se toma en las condiciones de esta misma figura, este primer pico queda marcado como modelo, con lo que se elimina de los cálculos en las siguientes medidas.

Realizando ahora una medida con dos blancos, ilustrada en la figura 32, se puede ver como la mayor sensibilidad repercute haciendo que las medidas pasen de nada a saturación. Como es obvio, el nivel de señal es mucho más alto, y por tanto es posible ganar algunos centímetros en la detección.

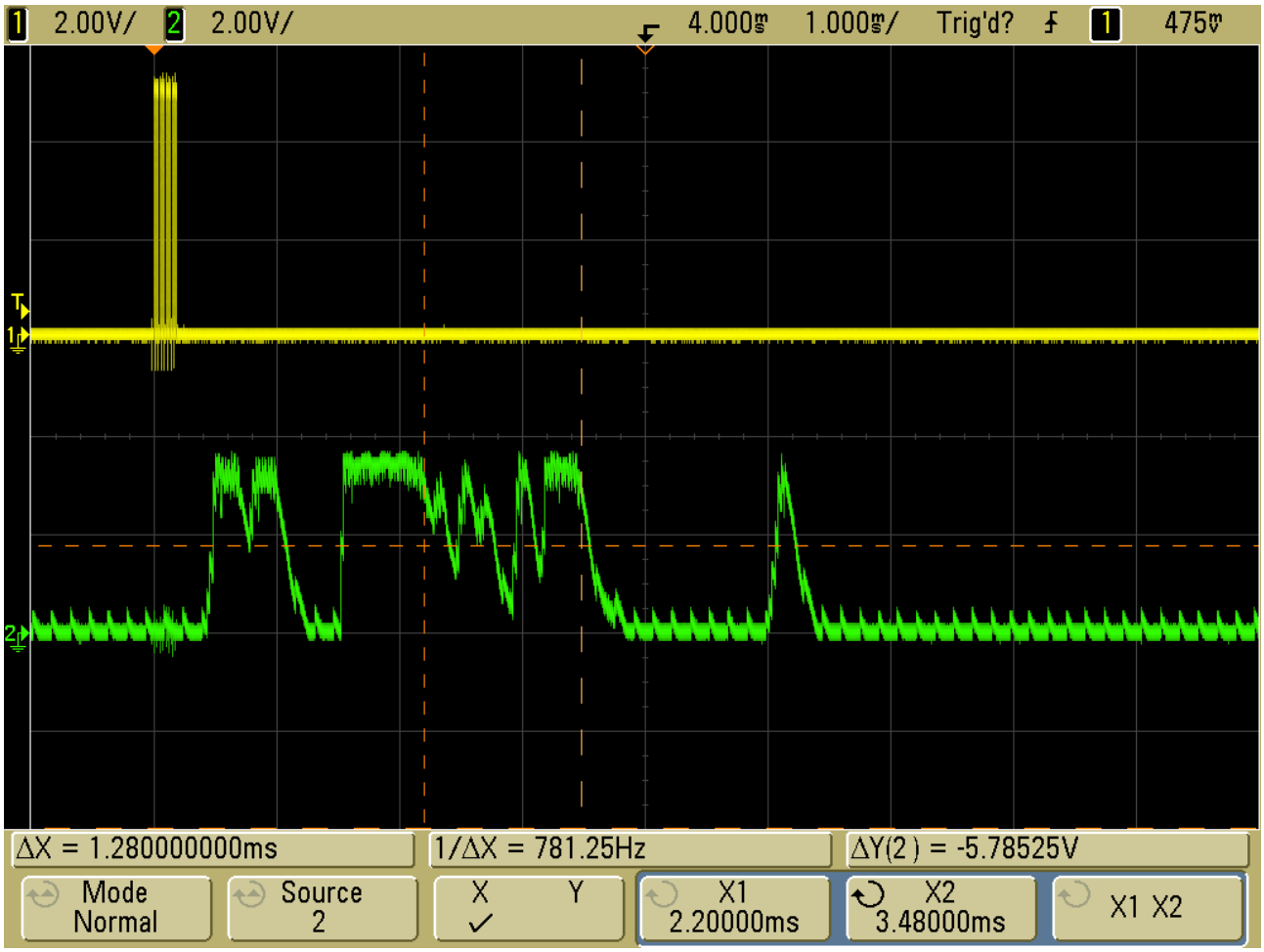

Fig. 32: Captura con dos blancos

El resultado obtenido en el PC en el caso de la figura 32 es el siguiente:

```
Peaks detected, node 7
0000 0000 0400 0004 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 - Peak #0 = 6720.0 us, distance = 2.30496 m.
    - Peak #1 = 8000.0 us, distance = 2.744 m.
```

Como se puede ver, el primer pico no se tiene en cuenta al formar parte del modelo, y solo se toman los dos siguientes. El último no se ve debido a que es demasiado estrecho para el detector y pasa como ruido. Igual que en el caso anterior, se tiene un desfase temporal que en el caso anterior, pero la posición relativa entre los picos (8000 µs – 6720 µs = 1280 µs, igual que entre los cursores de la figura 32) sigue siendo la correcta.

El desfase temporal presente en ambos casos es debido al tiempo de activación derivado de trabajar con el SRF02. Para corregirlo basta con modificar el orden en que se efectúan las llamadas en la programación. Como se comenta en el apartado 5.2.1 , antes se activaba en primer lugar el ADC y luego el pulso, pero como ahora el pulso tarda varios milisegundos en salir, se debe esperar a activar el ADC un tiempo después que se manda la orden de emitir el pulso tal que el tiempo medido en el osciloscopio y el medido en el PC coincidan.

5.4 Conclusiones de la implementación

A lo largo de este capítulo se ha realizado la implementación para los dos casos expuestos en la introducción: cuando el blanco puede portar un dispositivo identificador y cuando no puede portarlo.

El primer caso se basa en enviar un mensaje de sincronismo para avisar antes de enviar el pulso. Lo más importante resulta ser la necesidad de sincronizar perfectamente los instantes de emisión y captura del pulso de ultrasonidos, puesto que cualquier desviación influye muy negativamente en la precisión final de las medidas. Por ello es necesario interactuar a nivel bajo para asegurar que el funcionamiento interno del mecanismo de CSMA que lleva incorporado TinyOS no afecta.

El segundo caso envía y recibe el pulso de ultrasonidos desde el mismo nodo. De nuevo es muy importante la sincronización, pero en este caso no es necesario ser estricto con la radio, sino con el tiempo que se tarda entre que se pide que se envíe el pulso y el momento en que realmente se envía. En este caso se ha trabajado tanto con un circuito emisor propio como con un emisor comercial SRF02.

En ambos casos, una vez el nodo ha realizado los cálculos necesarios envía la información al PC, donde será procesada con el programa realizado en el apartado 4.2 .

6 Conclusiones

En primer lugar se ha hablado del estado del arte de las dos posibles tecnologías a utilizar, ultrasonidos y Ultra Wide Band. Pese a que la segunda es teóricamente más precisa y con mejores resultados, por motivos monetarios y de disponibilidad finalmente se utilizan emisores y receptores de ultrasonidos.

El procedimiento para encontrar los blancos consiste en enviar un pulso de ultrasonidos y medir el tiempo hasta la llegada, de manera similar al funcionamiento de un radar. Por ello es crítico encontrar el momento concreto en que se recibe un pulso, por lo que se investigan varias alternativas de detección de picos.

Después se explican las tecnologías utilizadas a lo largo del proyecto, lo que comprende el uso del nodo Crossbow TelosB, programado en TinyOS, así como de un emisor y receptor propios y el emisor comercial SRF02.

Antes de empezar la programación de los nodos para el trabajo final que realizan se prepara el PC para interactuar con ellos, instalándole todos los componentes software necesarios y creando un programa en Java que lee los datos del puerto serie o USB y extrae la información para mostrarla por pantalla. Asimismo, también se hace un programa preliminar para los nodos que captura la señal por el conversor analógico-digital y envía todos los datos, sin modificar, vía radio. Con esto se pueden ver gráficas en el ordenador de lo que se está recibiendo en un nodo.

El siguiente paso es programar los nodos para los dos casos de actuación propuestos para el proyecto:

- Para el caso *outdoor*, al ser posible dotar a cada blanco de un nodo, para medir la distancia en primer lugar se manda un paquete radio indicando la intención de transmitir el pulso de ultrasonidos. Para ello se tiene en cuenta el mecanismo de acceso al medio, puesto que se requiere de precisión en la localización. Eliminar este mecanismo es impensable, así que se adapta el instante de envío de los ultrasonidos al instante en que realmente se envía el paquete de sincronismo por radio.

  En este caso no tiene sentido promediar o realizar un modelo del escenario porque se parte de la premisa que los blancos estarán en perpetuo movimiento.

- Para el caso *indoor*, como los blancos no portan nodo, se opta por usar un sistema similar al de un sonar. El pulso de ultrasonidos rebota en los blancos y puede ser captado por el receptor de ultrasonidos del mismo nodo.

  Como en este caso los nodos están en posiciones estáticas, es posible promediar las medidas tomadas y crear un modelo del escenario antes de empezar el funcionamiento. El problema del promediado es que puede eliminar blancos móviles que sean correctos, pero a cambio elimina medidas espúrias. El modelo del escenario guarda la posición de los ecos fijos, causados por elementos estáticos, como sillas y mesas, y no los tiene en cuenta cuando toma las siguientes medidas.

Como conclusiones, a lo largo del proyecto se ha visto que presenta un buen comportamiento y el software es adaptable a multitud de situaciones, pero está muy limitado por los circuitos emisor y receptor de los que se disponga.

Con circuitos con poca sensibilidad el alcance es muy limitado, y aunque circuitos con más sensibilidad es más fácil detectar ecos lejanos, también es mucho más fácil que aparezcan espúrios o ecos inválidos al haber mayor nivel de ruido. Al aparecer estos ecos indeseados, no queda más remedio que intentar anularlos vía software, o bien mediante la adaptación del umbral de ruido del

algoritmo de detección de picos o bien mediante promediado.

Por esto se ha visto que la solución no puede ser completamente general, es decir, se deben ajustar los parámetros de ruido y alcance, tanto a nivel hardware, usando diferentes componentes como a nivel software, variando los parámetros del algoritmo según la situación concreta en que se vaya a utilizar para que los resultados sean lo mejor posibles.

En definitiva, se puede decir que se han cumplido los objetivos del proyecto propuestos en la introducción. Además, gran parte del trabajo se ha realizado en un grupo de trabajo, con lo que el trabajo en equipo ha sido esencial. Finalmente, en parte del proyecto se ha estado en estrecha comunicación con Vodafone, lo que ha aportado una interesante visión a título personal del ambiente de trabajo entre universidad y empresa.

7 Líneas futuras

Entre las posibles líneas por las que continuar el trabajo realizado en este proyecto se encuentran:

- Opcionalmente, se podría adaptar el algoritmo de detección de picos en uno más sencillo si se utiliza finalmente un circuito que sature la señal.

  Si en lugar de utilizar el circuito que amplifica la señal recibida se utiliza uno que posea un detector de pulsos hardware no es necesario utilizar un detector de picos tan complejo, así que podrían buscarse alternativas más sencillas.

- Utilizar Ultra Wide Band en lugar de ultrasonidos para ver las posibles mejoras que puede aportar. Entre las mejoras esperables se encuentran más precisión y más alcance, así como poder utilizar el pulso de UWB como radio, de manera que se podrían hacer medidas y comunicación entre nodos de manera indistinta.

  Este cambio de tecnología probablemente puede propiciar que se tengan que reajustar también los parámetros software, no sólo para adaptarse a los nuevos controladores, si no quizá también para mejorar los algoritmos al necesitar de una temporización mucho más precisa.

- Realizar un sistema central que controle todos los nodos y recopile y procese las medidas. Compilando las medidas de varios nodos es posible no sólo saber la distancia a la que está un elemento, si no su posición con respecto al sistema de referencia.

  Ahora mismo, la información obtenida es únicamente la distancia entre dos nodos en concreto, para el caso en que el objetivo porta dispositivo identificador, o la distancia entre el nodo y los blancos que detecta, en el caso en que el objetivo no porte dispositivo identificador. Si se procesa la información global del sistema en un computador central, es posible extraer la localización de los objetivos de manera más precisa y situada incluso en tres dimensiones y en el tiempo, lo que permitiría ver la evolución en la posición de los blancos.

  La creación del sistema central se prevé especialmente complicada para el caso de blancos que portan identificador propio, puesto que los emisores y receptores estarán en constante movimiento. Esto es muy probable que dificulte el posicionamiento global al ser más difícil o directamente imposible, según el caso concreto, tener un sistema de referencia estático.

- Siguiendo el punto anterior, mejorar el reconocimiento de blancos y elementos estáticos utilizando el sistema central. Al tener más información disponible es más fácil determinar qué elementos son los que se deben monitorizar y cuales no.

  Esto es importante para evitar que espurios o rebotes indeseados afecten al posicionamiento de los blancos que se quieren ubicar. Contra más compleja sea la situación en la que se encuentren los blancos, más crítico será poder filtrar los rebotes indeseados. Esto es muy difícil hacerlo con la información de sólo un nodo y por eso sería conveniente añadirlo a la programación del sistema central de control y recopilación de datos.

- Globalizar la solución. Esto podría pasar por conectarse con otro tipo de redes, como Internet, o usar otros dispositivos, como GPS, para combinar información global y local para obtener una visión panorámica de la situación de los blancos.

  Hoy en día existen mecanismos de localización con respecto al globo terrestre en general, y esta información se podría combinar con la información extraída del sistema de control para ofrecer un posicionamiento de los blancos con mucho más detalle.

Bibliografía y referencias

[1]: Nissanka B. Priyantha, Anit Chakraborty, Hari Balakrishnan, "The Cricket Location-Support System", *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pp. 32-43, August 2000

[2]: Harry S. Sameshima, Edward P. Katz, *Experiences with Cricket/Ultrasound Technology for 3-Dimensional Locationing within an Indoor Smart Environment*, http://www.cylab.cmu.edu/files/pdfs/mobility/Experiences_with_Crickets_MRC-TR-2009-01.pdf

[3]: Yu-Chee Tseng, *Active Bat: a 3D Location Device*, www.cs.nctu.edu.tw/~yctseng/WirelessNet2009-02/active-bat.ppt

[4]: A. Ward, A. Jones, A. Hopper, "A New Location Technique for the Active Office", *IEEE Personal Communications*, vol. 4, no. 5, pp. 42-47, October 1997

[5]: AT&T Laboratories Cambridge, *The Bat Ultrasonic Location System*, http://www.cl.cam.ac.uk/research/dtg/attarchive/bat/

[6]: Eric Foxlin, Michael Harrington, George Pfeifer, "Constellation: a Wide-Range Wireless Motion-Tracking System for Augmented Reality and Virtual Set Applications", *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 371-378, July 1998

[7]: Yasuhiro Fukuju, [et al.], "DOPLHIN: An Autonomous Indoor Positioning System in Ubiquitous Computing Environment", *IEEE Workshop on Software Technologies for Future Embedded Systems*, pp. 53-56, May 2003

[8]: Andreas Savvides, Chih-Chieh Han, Mani B. Srivastava, "Dynamic Fine-Grained Localization in Ad-Hoc Networks of Sensors", *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pp. 166-179, July 2001

[9]: Andreas Savvides, Mani B. Srivastava, "A Distributed Computation Platform for Wireless Embedded Sensing", *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 220-225, 2002

[10]: Michael Ronan McCarthy, *Narrowband Ultrasonic Positioning for Wearable Computers*, 2007, url: http://www.cs.bris.ac.uk/Publications/Papers/2000430.pdf

[11]: Michael McCarthy, Henk Muller, *Positioning with Independent Ultrasonic Beacons*, http://www.cs.bris.ac.uk/Publications/Papers/2000430.pdf

[12]: Mike Hazas, Andy Hopper, "Broadband Ultrasonic Location Systems for Improved Indoor Positioning", *IEEE Transaction on Mobile Computing*, vol. 5, no. 5, pp. 536-547, 2006

[13]: Decawave, *ScenSor Precision Asset Location Technical Data*, http://www.decawave.com/downloads.html

[14]: Ubisense, *Ubisense Precise Location*, http://www.ubisense.net/pdf/fact-sheets/products/software/Precise-Location-EN090624.pdf

[15]: Dongwoo Kang [et al.], "A Simple Asynchronous UWB Position Location Algorithm Based On Single Round-Trip Transmission", *The 8th International Conference on Advanced Communication Technology*, pp. , February 2006

[16]: Omar Abdul-Latif, Peter Shepherd, Stephen Pennock, "TDOA/AOA Data Fusion for Enhancing Positioning in an Ultra Wideband System", *IEEE International Conference on Signal Processing and Communications*, pp. 1531-1534, 2007

[17]: Valentin T. Jordanov, Dave L. Hall, Mat. Kastner, "Digital Peak Detector with Noise Threshold", *IEEE Nuclear Science Symposium*, pp. 140-142, November 2002

[18]: Valentin Jordanov, Glenn F. Knoll, "Digital Pulse Processor Using A Moving Average Technique", *IEEE Transactions on Nuclear Science*, vol. 40, no. 4, pp. 764-769, August 1993

[19]: K.C. Ho, Y.T. Chan, R.J. Inkol, "Pulse Arrival Time Estimation Based on Pulse Sample Ratios", *IEEE Proceedings, Radar, Sonar Navigation*, vol. 142, no. 4, pp. 153-157, August 1995

[20]: Crossbow, *TelosB Datasheet*, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf

[21]: Philip Levis, *TinyOS Programming*, http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf

[22]: David Gay, Philip Levis, David Culler, Eric Brewer, *nesC 1.1 Language Reference Manual*, http://nescc.sourceforge.net/papers/nesc-ref.pdf

[23]: Kevin Klues, [et al.], "Integrating Concurrency Control and Energy Management in Device Drivers", *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 251-264, October 2007

[24]: Devantech, *Devantech SRF02 Sensor*, http://www.acroname.com/robotics/parts/R287-SRF02.html

[25]: Ubuntu Linux Operating System, www.ubuntu.com

[26]: VMWare Server, www.vmware.com

[27]: XubunTOS Operating System, Xubuntu + TinyOS, http://toilers.mines.edu/Public/XubunTOS

[28]: Vim, Vi iMproved, www.vim.org

[29]: NetBeans IDE 6.7, www.netbeans.org

[30]: JFreeChart, Java Charts Library, www.jfree.org/jfreechart

8 Anexos

8.1 Código Java utilizado para el entorno gráfico

8.1.1 BitManager.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package usReceiver;

/**
 *  Class to work with bit values more easily.
 *
 * @author ivan
 */
public class BitManager {

    // Number of bits per array position this BitManager will work with.
    int numBits;
    public static int DEFAULT_NUM_BITS = 16;

    public BitManager(int numBits)
    {
        this.numBits = numBits;
    }

    public BitManager()
    {
        this.numBits = DEFAULT_NUM_BITS;
    }

    public int bitmask(int b)
    {
        //#define BITMASK(b) (1 << ((b) % NUM_BITS))
        return (1 << ((b) % numBits));
    }

    public int bitslot(int b)
    {
        //#define BITSLOT(b) ((b) / NUM_BITS)
        return (int)(Math.floor(b/numBits));
    }

    public int[] bitset(int[] a, int b)
    {
        //#define BITSET(a, b) ((a)[BITSLOT(b)] |= BITMASK(b))
        a[bitslot(b)] |= bitmask(b);
        return a;
    }

    public int[] bitclear(int[] a, int b)
    {
        //#define BITCLEAR(a, b) ((a)[BITSLOT(b)] &= ~BITMASK(b))
        a[bitslot(b)] &= ~bitmask(b);
```

```
        return a;
    }

    public Boolean bittest(int[] a, int b)
    {
        //#define BITTEST(a, b) ((a)[BITSLOT(b)] & BITMASK(b))
        return ((a[bitslot(b)] & bitmask(b)) != 0);
    }

    public int bitnslots(int nb)
    {
        //#define BITNSLOTS(nb) ((nb + NUM_BITS - 1) / NUM_BITS)
        return (int)Math.floor((nb + numBits - 1)/numBits);
    }

}
```

8.1.2 PeaksPrinter.java

```
package usReceiver;

import java.util.*;

import net.tinyos.message.*;
import net.tinyos.packet.*;
import net.tinyos.util.*;

public class PeaksPrinter implements net.tinyos.message.MessageListener
{
    private MoteIF moteIF;
    protected PeakDetMsg lastMsg = new PeakDetMsg();
    protected int msgIndex = 0;

    /** Which value means there is no peak */
    public static final int NO_PEAK = 65535;


    public PeaksPrinter(String source) throws Exception
    {
        if (source != null)
        {
            moteIF = new MoteIF(BuildSource.makePhoenix(source,
                        PrintStreamMessenger.err));
        }
        else
        {
            moteIF = new
MoteIF(BuildSource.makePhoenix(PrintStreamMessenger.err));
        }
    }

    public void messageReceived(int to, Message message)
    {
        PeakDetMsg msg = new PeakDetMsg(message, 0);
        System.out.println("Peaks detected, node " + msg.get_nodeid());
        int[] peaks = msg.get_peaks();
        int numBits = PeakDetMsg.elementSizeBits_peaks();
        BitManager bm = new BitManager(numBits);
        int k = 0;
        for (int i = 0; i < PeakDetMsg.numElements_peaks(); i++)
```

```
        {
            String hex = Integer.toHexString(peaks[i]);
            //
            for (int j = 0; j < (numBits/4-hex.length()); j++)
            {
                System.out.print("0");
            }
            System.out.print(hex + " ");
        }
        System.out.println();
        for (int i = 0; i < PeakDetMsg.numElements_peaks(); i++)
        {
            // Only if there is a peak on this position, test which of
            // the samples is.
            if (peaks[i] != 0)
            {
                for (int j = 0; j < numBits; j++)
                {
                    if (bm.bittest(peaks, (j+i*numBits)))
                    {
                        System.out.println(" - Peak #" + k + " = " +
                                (i*numBits+j)*Readings.SAMPLING_PERIOD_US +
                                " us, distance = " +
                                (i*numBits+j)*Readings.SAMPLING_PERIOD_US*
                                Readings.SOUND_SPEED/1000000 +
                                " m."); //Divide by 10^6
                        k++;
                    }
                }
            }
        }
        /**
         *
         * When using a peak for every position
        for (int i = 0; i < msg.numElements_peaks(); i++)
        {
            if (peaks[i] != NO_PEAK)
            {
                System.out.println(" - Peak #" + i + ": "+ peaks[i] +" = " +
                        peaks[i]*Readings.SAMPLING_PERIOD_US + " us, distance
= " +
                        peaks[i]*Readings.SAMPLING_PERIOD_US*
                        Readings.SOUND_SPEED/1000000 + " m."); //Divide by
10^6
            }
        }
         */
    }

    public void start()
    {
    }

    private void addMsgType(Message msg)
    {
        moteIF.registerListener(msg, this);
    }

    private static void usage() {
        System.err.println("usage: ReadingsPrinter [-comm <source>] message-
class [message-class ...]");
```

```
}

public static void main(String[] args) throws Exception
{
    String source = null;
    Vector<Message> v = new Vector<Message>();
    if (args.length > 0)
    {
        for (int i = 0; i < args.length; i++)
        {
            if (args[i].equals("-comm"))
            {
                source = args[++i];
            }
            else
            {
                /*
                String className = args[i];
                try
                {
                    Class c = Class.forName(className);
                    Object packet = c.newInstance();
                    Message msg = (Message)packet;
                    if (msg.amType() < 0)
                    {
                        System.err.println(className +
                                " does not have an AM type - ignored");
                    }
                    else
                    {
                        v.addElement(msg);
                    }
                }
                catch (Exception e)
                {
                    System.err.println(e);
                }*/
            }
        }
    }
    else if (args.length != 0)
    {
        usage();
        System.exit(1);
    }

    PeaksPrinter pp = new PeaksPrinter(source);
    /*
    Enumeration msgs = v.elements();
    while (msgs.hasMoreElements())
    {
        Message m = (Message)msgs.nextElement();
        pp.addMsgType(m);
    }
    */
    Class c = Class.forName("usReceiver.PeakDetMsg");
    Object packet = c.newInstance();
    Message msg = (Message)packet;
    pp.addMsgType(msg);
    pp.start();
}
```

```
}
```

8.1.3 PeaksProcesser.java

```
package usReceiver;

import java.util.*;

import net.tinyos.message.*;
import net.tinyos.packet.*;
import net.tinyos.util.*;
import org.jfree.data.xy.DefaultXYDataset;

public class PeaksProcesser implements net.tinyos.message.MessageListener
{

    private MoteIF moteIF;
    protected DefaultXYDataset dataset = new DefaultXYDataset();
    /** Boolean value indicating if new messages are being processed */
    protected boolean listen = false;
    protected boolean waitForCompleteMsg = true;

    /**
     * Gets the current dataset the chart is using.
     *
     * @return
     * dataset
     */
    public DefaultXYDataset getDataset()
    {
        return dataset;
    }

    /**
     * Tells if new messages are being processed.
     *
     * @return
     * true if messages are processed.
     */
    public boolean isListening()
    {
        return listen;
    }

    /**
     * Start listening to new messages.
     * Waits until last message array finishes to get a full one.
     */
    public void startListening()
    {
        this.waitForCompleteMsg = true;
    }

    /**
     * Stop listening to new messages.
     */
    public void stopListening()
    {
        this.listen = false;
    }
```

```
/**
 * Creates a new instance of ReadingsProcesser.
 * @param source
 * @throws Exception
 */
public PeaksProcesser(String source) throws Exception
{
    if (source != null)
    {
        moteIF = new MoteIF(BuildSource.makePhoenix(source,
                    PrintStreamMessenger.err));
    }
    else
    {
        moteIF = new MoteIF(BuildSource.
                makePhoenix(PrintStreamMessenger.err));
    }
}

/**
 * When a message is received, update the correct values and notify the
 * plotting code to redo the graphics.
 *
 * @param to
 * The id of the node the message has been sent to.
 * @param message
 * The message received.
 */
public void messageReceived(int to, Message message)
{
}

/**
 * Clears msgList so new readings can be obtained.
 */
public void clearMsgList()
{
}

/**
 * Prints the message with the given index on the chart.
 *
 * @param index
 * Must be on 1-base (the lowest index must be 1)
 */
public void showMsg(int index)
{
}


public void start()
{
}


public void addMsgType(Message msg)
{
    moteIF.registerListener(msg, this);
}
```

```
    private static void usage() {
        System.err.println("usage: ReadingsPrinter [-comm <source>] message-
class [message-class ...]");
    }

    public static void main(String[] args) throws Exception
    {
        String source = null;
        Vector<Message> v = new Vector<Message>();
        if (args.length > 0)
        {
            for (int i = 0; i < args.length; i++)
            {
                if (args[i].equals("-comm"))
                {
                    source = args[++i];
                }
            }
        }
        else if (args.length != 0)
        {
            usage();
            System.exit(1);
        }

        PeaksProcesser rp = new PeaksProcesser(source);
        /* usReceiver.PeakDetMsg must be manually updated every time
         * the tinyos code is recompiled. It must be overwritten, and then
         * <code>package usReceiver</code>
         * must be added to its code. Then recompile and rerun.
         */
        Class c = Class.forName("usReceiver.PeakDetMsg");
        Object packet = c.newInstance();
        Message msg = (Message)packet;
        rp.addMsgType(msg);
        rp.start();
    }

}
```

8.1.4 ReadingsPrinter.java

```
package usReceiver;

import java.util.*;

import net.tinyos.message.*;
import net.tinyos.packet.*;
import net.tinyos.util.*;

public class ReadingsPrinter implements net.tinyos.message.MessageListener
{
    private MoteIF moteIF;
    protected UsReceiverMsg modelMsg;
    protected UsReceiverMsg[] lastMsgs = new UsReceiverMsg[Readings.MAX_MSG];
    protected int msgIndex = 0;


    public ReadingsPrinter(String source) throws Exception
    {
        if (source != null)
```

```
        {
            moteIF = new MoteIF(BuildSource.makePhoenix(source,
                        PrintStreamMessenger.err));
        }
        else
        {
            moteIF = new
MoteIF(BuildSource.makePhoenix(PrintStreamMessenger.err));
        }
    }

    public void messageReceived(int to, Message message)
    {
        /*long t = System.currentTimeMillis();
        System.out.print("" + t + ": ");
        UsReceiverMsg msg = new UsReceiverMsg(message, 0);
        System.out.println(msg.get_nodeid());
        System.out.println(msg.get_flags());
        System.out.println("Is model?: " + Readings.isModel(msg));*/
        UsReceiverMsg msg = new UsReceiverMsg(message, 0);
        if (Readings.isModel(msg))
        {
            modelMsg = msg;
        }
        else
        {
            if (msgIndex < Readings.MAX_MSG)
            {
                lastMsgs[msgIndex] = msg;
                msgIndex++;
                //if (msgIndex >= Readings.MAX_MSG)
                //    msgIndex = 0;
            }

        }
        if (msgIndex == Readings.MAX_MSG)
        {
            System.out.println( Readings.MAX_MSG + " messages received.");
            System.out.println("Model: ");
            if (modelMsg != null)
                System.out.println(modelMsg.toString());
            else
                System.out.println(" no model available ");
            for (int i = 0; i < Readings.MAX_MSG; i++)
            {
                System.out.print( i + ": " + lastMsgs[i].toString() + "");
                System.out.println("Value: " +
lastMsgs[i].getElement_data(0));
            }
            msgIndex = 0;
        }
        /*
        if (modelMsg != null)
            System.out.println("Has changed?: " +
         Readings.hasChanged(modelMsg, msg));
         * */
    }

    public void start()
    {
    }
```

```
    private void addMsgType(Message msg)
    {
        moteIF.registerListener(msg, this);
    }

    private static void usage() {
        System.err.println("usage: ReadingsPrinter [-comm <source>] message-
class [message-class ...]");
    }

    public static void main(String[] args) throws Exception
    {
        String source = null;
        Vector<Message> v = new Vector<Message>();
        if (args.length > 0)
        {
            for (int i = 0; i < args.length; i++)
            {
                if (args[i].equals("-comm"))
                {
                    source = args[++i];
                }
                else
                {
                    /*
                    String className = args[i];
                    try
                    {
                        Class c = Class.forName(className);
                        Object packet = c.newInstance();
                        Message msg = (Message)packet;
                        if (msg.amType() < 0)
                        {
                            System.err.println(className +
                                    " does not have an AM type - ignored");
                        }
                        else
                        {
                            v.addElement(msg);
                        }
                    }
                    catch (Exception e)
                    {
                        System.err.println(e);
                    }*/
                }
            }
        }
        else if (args.length != 0)
        {
            usage();
            System.exit(1);
        }

        ReadingsPrinter rp = new ReadingsPrinter(source);
        /*
        Enumeration msgs = v.elements();
        while (msgs.hasMoreElements())
        {
            Message m = (Message)msgs.nextElement();
```

```
            rp.addMsgType(m);
        }
        */
        Class c = Class.forName("usReceiver.UsReceiverMsg");
        Object packet = c.newInstance();
        Message msg = (Message)packet;
        rp.addMsgType(msg);
        rp.start();
    }

}
```

8.1.5 ReadingsProcesser.java

```
package usReceiver;

import java.util.*;

import net.tinyos.message.*;
import net.tinyos.packet.*;
import net.tinyos.util.*;
import org.jfree.data.xy.DefaultXYDataset;

public class ReadingsProcesser implements net.tinyos.message.MessageListener
{

    private MoteIF moteIF;
    protected ArrayList<UsReceiverMsg> modelMsg =
            new ArrayList<UsReceiverMsg>();
    protected ArrayList<UsReceiverMsg>[] msgList =
            new ArrayList[Readings.MAX_MSG];
    protected UsReceiverMsg[] lastMsgs =
            new UsReceiverMsg[Readings.MAX_MSG];
    protected int msgIndex = 0;
    protected int blockIndex = 0;
    protected DefaultXYDataset dataset = new DefaultXYDataset();
    /** Boolean value indicating if new messages are being processed */
    protected boolean listen = false;
    protected boolean waitForCompleteMsg = true;

    protected int samples[] = new int[Readings.TOTAL_SAMPLES];

    /**
     * Gets the current dataset the chart is using.
     *
     * @return
     * dataset
     */
    public DefaultXYDataset getDataset()
    {
        return dataset;
    }

    /**
     * Tells if new messages are being processed.
     *
     * @return
     * true if messages are processed.
     */
    public boolean isListening()
    {
```

```
    return listen;
}

/**
 * Start listening to new messages.
 * Waits until last message array finishes to get a full one.
 */
public void startListening()
{
    this.waitForCompleteMsg = true;
}

/**
 * Stop listening to new messages.
 */
public void stopListening()
{
    this.listen = false;
}

/**
 * Creates a new instance of ReadingsProcesser.
 * @param source
 * @throws Exception
 */
public ReadingsProcesser(String source) throws Exception
{
    if (source != null)
    {
        moteIF = new MoteIF(BuildSource.makePhoenix(source,
                    PrintStreamMessenger.err));
    }
    else
    {
        moteIF = new MoteIF(BuildSource.
                makePhoenix(PrintStreamMessenger.err));
    }
    for (int i = 0; i < Readings.MAX_MSG; i++)
    {
        msgList[i] = new ArrayList<UsReceiverMsg>();
    }
}

/**
 * When a message is received, update the correct values and notify the
 * plotting code to redo the graphics.
 *
 * @param to
 * The id of the node the message has been sent to.
 * @param message
 * The message received.
 */
public void messageReceived(int to, Message message)
{
    if (message.amType() == UsReceiverMsg.AM_TYPE)
    {
        UsReceiverMsg msg = new UsReceiverMsg(message, 0);

        if (isListening())
        {
            System.out.print(msg);
```

```
                if (Readings.isModel(msg))
                {
                    modelMsg.add(msg);
                    System.out.println("model added");
                }
                else
                {
                    msgList[msgIndex].add(msg);
                    System.out.println("message added");
                }
                if (Readings.isLastMsg(msg))
                {
                    // When its the last message on a string, draw it
                    if (Readings.isModel(msg))
                    {
                        Readings.insertIntoDataset(dataset,
                                modelMsg, Readings.MODEL);
                    }
                    else
                    {
                        msgIndex++;
                        if (msgIndex >= Readings.MAX_MSG)
                        {
                            msgIndex = 0;
                            Readings.insertIntoDataset(dataset,
                                    msgList[Readings.MAX_MSG-1],
Readings.MSG);
                            stopListening();
                        }
                    }
                }
            }
            else if (waitForCompleteMsg)
            {
                if (Readings.isLastMsg(msg))
                {
                    listen = true;
                    waitForCompleteMsg = false;
                }
            }
        }
    }

    /**
     * Clears msgList so new readings can be obtained.
     */
    public void clearMsgList()
    {
        for (int i = 0; i < msgList.length; i++)
        {
           this.msgList[i].clear();
        }
    }

    /**
     * Prints the message with the given index on the chart.
     *
     * @param index
     * Must be on 1-base (the lowest index must be 1)
     */
    public void showMsg(int index)
```

```
    {
        // The index is converted from 1-base to 0-base
        index--;
        if (index >= msgIndex)
        {
            // The chart updates automatically every time its associated
            // XYDataset changes.
            this.dataset = Readings.insertIntoDataset(dataset, msgList[index],
                    Readings.MSG);
        }
    }


    public void start()
    {
    }


    public void addMsgType(Message msg)
    {
        moteIF.registerListener(msg, this);
    }

    private static void usage() {
        System.err.println("usage: ReadingsPrinter [-comm <source>] message-
class [message-class ...]");
    }

    public static void main(String[] args) throws Exception
    {
        String source = null;
        Vector<Message> v = new Vector<Message>();
        if (args.length > 0)
        {
            for (int i = 0; i < args.length; i++)
            {
                if (args[i].equals("-comm"))
                {
                    source = args[++i];
                }
            }
        }
        else if (args.length != 0)
        {
            usage();
            System.exit(1);
        }

        ReadingsProcesser rp = new ReadingsProcesser(source);
        /* usReceiver.UsReceiverMsg must be manually updated every time
         * the tinyos code is recompiled. It must be overwritten, and then
         * <code>package usReceiver</code>
         * must be added to its code. Then recompile and rerun.
         */
        Class c = Class.forName("usReceiver.UsReceiverMsg");
        Object packet = c.newInstance();
        Message msg = (Message)packet;
        rp.addMsgType(msg);
        rp.start();
    }
```

```
}
```

8.1.6 PeaksViewer.java

```
package usReceiver.gui;

/**
 *
 * @author ivan
 */
public class PeaksViewer extends javax.swing.JFrame {

    /** Creates new form PeaksViewer */
    public PeaksViewer() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG)
            .addGap(0, 400, Short.MAX_VALUE)
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG)
            .addGap(0, 300, Short.MAX_VALUE)
        );

        pack();
    }// </editor-fold>//GEN-END:initComponents

    /**
    * @param args the command line arguments
    */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new PeaksViewer().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    // End of variables declaration//GEN-END:variables

}
```

8.1.7 ReadingsViewer.java

```
package usReceiver.gui;

import java.io.File;
import java.io.IOException;
import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;
import net.tinyos.message.Message;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import usReceiver.Readings;
import usReceiver.ReadingsProcesser;

/**
 *
 * @author Iván Tomás
 */
public class ReadingsViewer extends javax.swing.JFrame {

    /** The ReadingsProcesser where messages are being caught and updated */
    public static ReadingsProcesser rp;

    /** chart to show the measures */
    private static JFreeChart chart;
    private static ChartPanel cp;

    /** Creates new form ReadingsViewer */
    public ReadingsViewer() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {
        bindingGroup = new org.jdesktop.beansbinding.BindingGroup();

        jFileDialog = new javax.swing.JDialog();
        jfcPNGFile = new javax.swing.JFileChooser();
        jlblFileAndResolution = new javax.swing.JLabel();
        jtfPNGHeight = new javax.swing.JTextField();
        jlblX = new javax.swing.JLabel();
        jtfPNGWidth = new javax.swing.JTextField();
        jlblResolution = new javax.swing.JLabel();
        jSeparator1 = new javax.swing.JSeparator();
        jsldSelectMessage = new javax.swing.JSlider();
        jbtnGetNew = new javax.swing.JButton();
        jlblMessageNumber = new javax.swing.JLabel();
        jifChart = new javax.swing.JInternalFrame();
        jbtnSaveAsPng = new javax.swing.JButton();
```

```
        jFileDialog.setTitle("Select filename and resolution");

        jlblFileAndResolution.setText("Filename (*.png)");

        jtfPNGHeight.setText("640");

        jlblX.setText("x");

        jtfPNGWidth.setText("480");

        jlblResolution.setText("Resolution");

        jSeparator1.setOrientation(javax.swing.SwingConstants.VERTICAL);

        javax.swing.GroupLayout jFileDialogLayout = new
javax.swing.GroupLayout(jFileDialog.getContentPane());
        jFileDialog.getContentPane().setLayout(jFileDialogLayout);
        jFileDialogLayout.setHorizontalGroup(
            jFileDialogLayout.createParallelGroup(javax.swing.GroupLayout.Alig
nment.LEADING)
            .addGroup(jFileDialogLayout.createSequentialGroup()
                .addContainerGap()
                .addGroup(jFileDialogLayout.createParallelGroup(javax.swing.Gr
oupLayout.Alignment.LEADING)
                    .addComponent(jfcPNGFile,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(jlblFileAndResolution))
                .addGap(12, 12, 12)
                .addComponent(jSeparator1,
javax.swing.GroupLayout.PREFERRED_SIZE, 10,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
RELATED)
                .addGroup(jFileDialogLayout.createParallelGroup(javax.swing.Gr
oupLayout.Alignment.LEADING)
                    .addComponent(jlblResolution)
                    .addGroup(jFileDialogLayout.createSequentialGroup()
                        .addComponent(jtfPNGHeight,
javax.swing.GroupLayout.PREFERRED_SIZE, 57,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlac
ement.RELATED)
                        .addComponent(jlblX)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlac
ement.RELATED)
                        .addComponent(jtfPNGWidth,
javax.swing.GroupLayout.PREFERRED_SIZE, 58,
javax.swing.GroupLayout.PREFERRED_SIZE)))
                .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        );
        jFileDialogLayout.setVerticalGroup(
            jFileDialogLayout.createParallelGroup(javax.swing.GroupLayout.Alig
nment.LEADING)
            .addGroup(jFileDialogLayout.createSequentialGroup()
                .addGroup(jFileDialogLayout.createParallelGroup(javax.swing.Gr
oupLayout.Alignment.LEADING)
                    .addGroup(jFileDialogLayout.createSequentialGroup()
                        .addContainerGap()
```

```
                        .addGroup(jFileDialogLayout.createParallelGroup(javax.
swing.GroupLayout.Alignment.LEADING)
                            .addComponent(jSeparator1,
javax.swing.GroupLayout.Alignment.TRAILING,
javax.swing.GroupLayout.DEFAULT_SIZE, 457, Short.MAX_VALUE)
                            .addGroup(jFileDialogLayout.createSequentialGroup(
)
                                .addComponent(jlblFileAndResolution)
                                .addPreferredGap(javax.swing.LayoutStyle.Compo
nentPlacement.UNRELATED)
                                .addComponent(jfcPNGFile,
javax.swing.GroupLayout.DEFAULT_SIZE, 430, Short.MAX_VALUE))))
                    .addGroup(jFileDialogLayout.createSequentialGroup()
                        .addGap(204, 204, 204)
                        .addComponent(jlblResolution)
                        .addGap(18, 18, 18)
                        .addGroup(jFileDialogLayout.createParallelGroup(javax.
swing.GroupLayout.Alignment.BASELINE)
                            .addComponent(jtfPNGHeight,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addComponent(jlblX)
                            .addComponent(jtfPNGWidth,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))))
                .addContainerGap())
        );

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        jsldSelectMessage.setMaximum(Readings.MAX_MSG);
        jsldSelectMessage.setMinimum(1);
        jsldSelectMessage.setPaintLabels(true);
        jsldSelectMessage.setPaintTicks(true);
        jsldSelectMessage.setSnapToTicks(true);
        jsldSelectMessage.setValue(Readings.MAX_MSG);
        jsldSelectMessage.addChangeListener(new
javax.swing.event.ChangeListener() {
            public void stateChanged(javax.swing.event.ChangeEvent evt) {
                jsldSelectMessageStateChanged(evt);
            }
        });

        jbtnGetNew.setText("Get new set");
        jbtnGetNew.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent evt) {
                jbtnGetNewMouseClicked(evt);
            }
        });

        org.jdesktop.beansbinding.Binding binding =
org.jdesktop.beansbinding.Bindings.createAutoBinding(org.jdesktop.beansbinding
.AutoBinding.UpdateStrategy.READ_WRITE, jsldSelectMessage,
org.jdesktop.beansbinding.ELProperty.create("${value}"), jlblMessageNumber,
org.jdesktop.beansbinding.BeanProperty.create("text"));
        bindingGroup.addBinding(binding);

        jifChart.setBorder(javax.swing.BorderFactory.createEtchedBorder(new
java.awt.Color(153, 204, 255), new java.awt.Color(102, 102, 102)));
        jifChart.setRequestFocusEnabled(false);
        jifChart.setVisible(true);
```

```
        javax.swing.GroupLayout jifChartLayout = new
javax.swing.GroupLayout(jifChart.getContentPane());
        jifChart.getContentPane().setLayout(jifChartLayout);
        jifChartLayout.setHorizontalGroup(
            jifChartLayout.createParallelGroup(javax.swing.GroupLayout.Alignme
nt.LEADING)
            .addGap(0, 676, Short.MAX_VALUE)
        );
        jifChartLayout.setVerticalGroup(
            jifChartLayout.createParallelGroup(javax.swing.GroupLayout.Alignme
nt.LEADING)
            .addGap(0, 347, Short.MAX_VALUE)
        );

        jbtnSaveAsPng.setText("Save as PNG");
        jbtnSaveAsPng.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent evt) {
                jbtnSaveAsPngMouseClicked(evt);
            }
        });

        javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG)
            .addGroup(layout.createSequentialGroup()
                .addContainerGap()
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.A
lignment.LEADING)
                    .addGroup(layout.createSequentialGroup()
                        .addGroup(layout.createParallelGroup(javax.swing.Group
Layout.Alignment.LEADING)
                            .addGroup(javax.swing.GroupLayout.Alignment.TRAILI
NG, layout.createSequentialGroup()
                                .addComponent(jsldSelectMessage,
javax.swing.GroupLayout.DEFAULT_SIZE, 569, Short.MAX_VALUE)
                                .addPreferredGap(javax.swing.LayoutStyle.Compo
nentPlacement.RELATED)
                                .addComponent(jlblMessageNumber)
                                .addGap(6, 6, 6))
                            .addGroup(layout.createSequentialGroup()
                                .addComponent(jbtnGetNew)
                                .addGap(18, 18, 18)
                                .addComponent(jbtnSaveAsPng)))
                        .addGap(89, 89, 89))
                    .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()
                        .addComponent(jifChart)
                        .addContainerGap())))
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADI
NG)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()
                .addComponent(jifChart)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RE
LATED)
```

```
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.A
lignment.LEADING)
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jsldSelectMessage,
javax.swing.GroupLayout.PREFERRED_SIZE, 45,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlac
ement.RELATED)
                        .addGroup(layout.createParallelGroup(javax.swing.Group
Layout.Alignment.BASELINE)
                            .addComponent(jbtnGetNew,
javax.swing.GroupLayout.PREFERRED_SIZE, 25,
javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addComponent(jbtnSaveAsPng)))
                    .addComponent(jlblMessageNumber))
                .addContainerGap())
        );

        bindingGroup.bind();

        pack();
    }// </editor-fold>//GEN-END:initComponents

    private void jsldSelectMessageStateChanged(javax.swing.event.ChangeEvent
evt) {//GEN-FIRST:event_jsldSelectMessageStateChanged
        // Changes the dataset to match the index selected with the slider.
        rp.showMsg(jsldSelectMessage.getValue());
    }//GEN-LAST:event_jsldSelectMessageStateChanged

    private void jbtnGetNewMouseClicked(java.awt.event.MouseEvent evt) {//GEN-
FIRST:event_jbtnGetNewMouseClicked
        // Starts listening again for new messages.
        rp.clearMsgList();
        rp.startListening();
    }//GEN-LAST:event_jbtnGetNewMouseClicked

    private void jbtnSaveAsPngMouseClicked(java.awt.event.MouseEvent evt)
{//GEN-FIRST:event_jbtnSaveAsPngMouseClicked
        //jFileDialog.requestFocus();
        File file = new File("chart.png");
        try {
            ChartUtilities.saveChartAsPNG(file, chart, 640, 480);
        } catch (IOException ex) {
            Logger.getLogger(ReadingsViewer.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }//GEN-LAST:event_jbtnSaveAsPngMouseClicked

    /**
     * Prints usage on the screen.
     */
    private static void usage() {
        System.err.println("usage: ReadingsPrinter [-comm <source>] message-
class [message-class ...]");
    }

    /**
     * Changes the value of jsldSelectMessage.
     *
     * @param value
     */
```

```
public static void setMessageSlider(int value)
{
    jsldSelectMessage.setValue(value);
}

/**
* @param args the command line arguments
*/
public static void main(String args[]) throws Exception{
    String source = null;
    Vector<Message> v = new Vector<Message>();
    if (args.length > 0)
    {
        for (int i = 0; i < args.length; i++)
        {
            if (args[i].equals("-comm"))
            {
                source = args[++i];
            }
        }
    }
    else if (args.length != 0)
    {
        usage();
        System.exit(1);
    }
    rp = new ReadingsProcesser(source);
    Class c = Class.forName("usReceiver.UsReceiverMsg");
    Object packet = c.newInstance();
    Message msg = (Message)packet;
    rp.addMsgType(msg);
    rp.start();

    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new ReadingsViewer().setVisible(true);
        }
    });

    // The first chart is drawn.
    // It will be automatically updated every time its XYDataset changes
    // due to JFreeChart's functionality.
    updateChart();

}

/**
 * Updates the chart shown at jifChart.
 */
public static void updateChart()
{
    // A XY Line chart is selected to show the values.
    chart = ChartFactory.createXYLineChart( //XYLineChart
            "Readings", //title
            "time[us]", //xAxisLabel
            "reading[V]", //yAxisLabel
            rp.getDataset(), //XYDataset
            PlotOrientation.VERTICAL,
            true, true, true); //legend, tooltips, urls
    // The panel is filled with the chart.
    cp = new ChartPanel(chart);
```

```
        jifChart.setContentPane(cp);

    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JDialog jFileDialog;
    private javax.swing.JSeparator jSeparator1;
    private javax.swing.JButton jbtnGetNew;
    private javax.swing.JButton jbtnSaveAsPng;
    private javax.swing.JFileChooser jfcPNGFile;
    private static javax.swing.JInternalFrame jifChart;
    private javax.swing.JLabel jlblFileAndResolution;
    private javax.swing.JLabel jlblMessageNumber;
    private javax.swing.JLabel jlblResolution;
    private javax.swing.JLabel jlblX;
    private static javax.swing.JSlider jsldSelectMessage;
    private javax.swing.JTextField jtfPNGHeight;
    private javax.swing.JTextField jtfPNGWidth;
    private org.jdesktop.beansbinding.BindingGroup bindingGroup;
    // End of variables declaration//GEN-END:variables

}
```

8.2 Código para emisor y receptor por separado

8.2.1 Emisor

8.2.1.1 UsReceiver.h

```
#ifndef USRECEIVER_H
#define USRECEIVER_H

enum{
    // Time between synchronizations
    TIMER_PERIOD_MILLI = 4000,
    // Conversor
    VOLTAGE_CONVERSION = 3/4096,
    // Sampling
    NUM_SAMPLES = 16,
    //MSG_SAMPLES = 96,
    TOTAL_SAMPLES = 576,
    //MSGS_TO_SEND = (TOTAL_SAMPLES/MSG_SAMPLES), //Must be integer
    JIFFIES = 160,
    SAMPLES_TO_TIME = (JIFFIES), //Find real formula
    NUM_PEAKS = 30,
    NO_PEAK = 0,
    // Threshold that will be considered noise.
    // To calculate a new threshold, divide desired voltage by
    // VOLTAGE_CONVERSION. For instance, for a threshold of 0.5 V,
    // this constant must be 0.5*4096/3 = 682.6666 => 683
    NOISE_THRESHOLD = 200, //173, //0.2V
    // for the peak detection algorithm
    TRACK_MAX = 1,
    TRACK_MIN = 0,
    // Minimum number of samples between succesful peak detections
    // If PEAK_MARGIN SAMPLES == 0, leave no margin
    PEAK_MARGIN_SAMPLES = 0,
    // Queues
    UART_QUEUE_LEN = 6,
    RADIO_QUEUE_LEN = 6,
    MAX_CLIENTS = 10,
    //flags
    IS_MODEL = 0x01,
    LAST_MSG = 0x02,
    SYNC_FLAG = 0x01,
    //messages constants
    AM_USRECEIVERMSG = 4,
    AM_SYNCMSG = 6,
    AM_PEAKDETMSG = 5,
};

typedef struct SyncMsg {
    uint16_t flag;
    uint16_t src;
} SyncMsg;

typedef nx_struct PeakDetMsg {
    nx_uint16_t nodeid;
    nx_uint16_t syncid;
    nx_uint16_t flags;
    nx_uint16_t peaks[NUM_PEAKS];
```

```
} PeakDetMsg;


#endif
```

8.2.1.2 UsReceiverAppC.nc

```
// Configuration file for UsReceiver.

#include <Timer.h>
#include "UsReceiver.h"
#include "printf.h"

configuration UsReceiverAppC{
}
implementation{
    components UsReceiverC as App;
    components MainC;
    components LedsC;
    components new TimerMilliC() as Timer0;

    // Analog channels
    components new Msp430Adc12ClientC() as AdcClient;

    // Radio
    components ActiveMessageC;
    // Information message
    //components new AMSenderC(AM_USRECEIVERMSG) as RadioSender;
    //components new AMReceiverC(AM_USRECEIVERMSG) as RadioReceiver;
    // Synchronization message
    //components new AMSenderC(AM_SYNCMSG) as SyncSender;
    //components new AMReceiverC(AM_SYNCMSG) as SyncReceiver;

    components new RadioAppC(AM_USRECEIVERMSG) as Radio;
    components new RadioAppC(AM_SYNCMSG) as Sync;
    components new RadioAppC(AM_PEAKDETMSG) as Peak;

    /** Wiring */

    App.Boot -> MainC.Boot;
    App.Timer0 -> Timer0;
    App.Leds -> LedsC;

    // Sensor wiring
    App.SensorResource -> AdcClient.Resource;
    App.ReadSensor -> AdcClient.Msp430Adc12SingleChannel;

    // Radio wiring
    App.RadioControl -> ActiveMessageC;
    // Sync wiring
    App.SyncSend -> Sync.Send[0]; //Only one client
    App.SyncPacket -> Sync.Packet;
    App.SyncReceive -> Sync.Receive[AM_SYNCMSG];
    App.SyncStdControl -> Sync.StdControl;
    App.SyncAMPacket -> ActiveMessageC;
    // Peak detection wiring
    App.PeakSend -> Peak.Send;
    App.PeakPacket -> Peak.Packet;
    App.PeakReceive -> Peak.Receive[AM_PEAKDETMSG];
    App.PeakStdControl -> Peak.StdControl;
```

```
    App.PeakAMPacket -> ActiveMessageC;

}
```

8.2.1.3 UsReceiverC.nc

```
/*                                                      tab:4
 * "Copyright (c) 2000-2005 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement
is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY
OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2005 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

#include <Timer.h>
#include "printf.h"
#include "UsReceiver.h"

module UsReceiverC
{
    uses
    {
        interface Boot;

        interface SplitControl as RadioControl;
        // Synchronization
        interface Receive as SyncReceive;
        interface Send as SyncSend;
        interface Packet as SyncPacket;
        interface AMPacket as SyncAMPacket;
        interface StdControl as SyncStdControl;
        // Peaks
        interface Receive as PeakReceive;
        interface Send as PeakSend[uint8_t client];
        interface Packet as PeakPacket;
        interface AMPacket as PeakAMPacket;
        interface StdControl as PeakStdControl;
```

```
        interface Msp430Adc12SingleChannel as ReadSensor;
        interface Resource as SensorResource;

        interface Timer<TMilli> as Timer0;
        interface Leds;

    }

}

implementation
{
    const msp430adc12_channel_config_t config = {
        INPUT_CHANNEL_A0,           //input channel
        REFERENCE_AVcc_AVss,        //reference voltage
        REFVOLT_LEVEL_NONE,         //reference voltage level
        SHT_SOURCE_ADC12OSC,        //clock source sample-hold-time
        SHT_CLOCK_DIV_1,            //clock divider sample-hold-time
        SAMPLE_HOLD_384_CYCLES,     //sample-hold-time
        SAMPCON_SOURCE_SMCLK,       //clock source sampcon signal
        SAMPCON_CLOCK_DIV_1         //clock divider sampcon
    };

    // Buffer to save temporary measures.
    uint16_t buffer[NUM_SAMPLES];
    // Measures that will be sent on the next packet
    //uint16_t lastReadings[MSG_SAMPLES];
    uint16_t totalReadings[TOTAL_SAMPLES];
    uint8_t readingsIndex;
    uint16_t totalIndex;
    // NodeID of the last synchronization received
    uint16_t lastSyncId;

    // Sync packet
    message_t spkt;
    // If sync is locked, do not take new measures
    bool syncLocked;
    uint8_t peakClient;
    SyncMsg *syncpkt;

    // Message containing the peak detection results
    PeakDetMsg *peakpkt;
    message_t ppkt[MAX_CLIENTS];

    void reportProblem()
    {
        call Leds.led0Toggle();
    }
    void reportSent()
    {
        call Leds.led1Toggle();
    }
    void reportReceived()
    {
        call Leds.led2Toggle();
    }

    /**
      Sets next peakClient for the new use
      */
```

```
    void nextPeakClient()
    {
        peakClient++;
        if (peakClient >= MAX_CLIENTS)
           peakClient = 0;
    }

    /**
      Prints peaks locations on the terminal
      */
    void printPeaks(nx_uint16_t lastPeaks[])
    {
#ifdef PRINT_PEAKS
        uint8_t i;
        printf("Peaks detected:\n");
        for (i = 0; i < NUM_PEAKS; i++)
        {
            printf("peak %u = %u, time = %u us\n", i, lastPeaks[i],
                    lastPeaks[i]*SAMPLES_TO_TIME);
        }
        printfflush();
#endif
    }


    /**
      Simulates a PREG register, a one edge register with enable,
      in order to detect peaks
      */
    int preg(bool enable, int in, int lastPreg)
    {
        if (enable)
        {
            if (in > lastPreg)
            {
                return in;
            }
            else
                return lastPreg;
        }
        else
        {
            if (in <= lastPreg)
            {
                return in;
            }
            else
                return lastPreg;
        }
    }


    /**
      Detects peaks on a sample buffer and saves their coordinates on
      lastPeaks.
      */
    void detectPeaks(nx_uint16_t lastPeaks[])
    {
        uint16_t i;
        uint8_t peakIndex = 0;
        int noise = NOISE_THRESHOLD;
```

```
        int pr = 0;
        bool enablePreg;
        int partial = 0;
        // The jump from TRACK_MAX to TRACK_MIN signifies a detected peak
        uint8_t lastOut = TRACK_MAX;
        uint8_t out = TRACK_MAX;
        uint8_t sign = 0;
        // Initialize margin so first samples are detectable
        uint8_t margin = PEAK_MARGIN_SAMPLES;
        for (i = 0; i < NUM_PEAKS; i++)
        {
            lastPeaks[i] = NO_PEAK;
        }
        for (i = 0; i < TOTAL_SAMPLES; i++)
        {
            lastOut = out;

            if (lastOut == TRACK_MIN)
                noise = NOISE_THRESHOLD;
            else
                noise = -NOISE_THRESHOLD;


            partial = totalReadings[i] - pr;
            //printf("(%d)", lastOut);
            //printf("%d ", partial);

            if (partial >= 0)
                sign = 0;
            else if (partial < 0)
                sign = 1;

            enablePreg = lastOut^sign;
            pr = preg(enablePreg, totalReadings[i],pr);

            if (partial > noise)
                out = TRACK_MAX;
            else
                out = TRACK_MIN;

            // If PEAK_MARGIN_SAMPLES samples have passed, enable detection
again
            // If PEAK_MARGIN SAMPLES == 0, leave no margin
            if (margin < PEAK_MARGIN_SAMPLES)
            {
                margin++;
            }
            else if ((out == TRACK_MIN)&&(lastOut == TRACK_MAX))
            {
                // Peak detected
#ifdef PRINT_PEAKS
                printf("Peak detected: %u, %u\n", totalReadings[i-1],
totalReadings[i]);
                printfflush();
#endif
                lastPeaks[peakIndex] = i;
                peakIndex++;
                // Reset margin
                margin = 0;
                // Once the first NUM_PEAKS are detected, exit the function
                if (peakIndex >= NUM_PEAKS)
```

```
                {
                    printPeaks(lastPeaks);
                    return;
                }
            }
        }
        printPeaks(lastPeaks);
    }


    /**
      Readies the sensor and calls getData() to start getting samples
      */
    void readySensor()
    {
        //First we must initialize the sensor.
        //Since it is MultipleRepeat, we must initialize it each time.
        error_t result = call ReadSensor.configureMultipleRepeat(&config,
                buffer,
                NUM_SAMPLES,
                JIFFIES);
        if (result == SUCCESS)
        {
            //If the initalization was correct, start reading data.
#ifdef DEBUG
            printf("Calling getData\n");
#endif
            call ReadSensor.getData();
        }
        else
            reportProblem();
    }


    /**
      Task to send a synchronization message so other motes wait for the US
      pulse.
      Uses syncpkt* to point to the payload of syncpkt
      */
    task void sendSyncPacket()
    {
        error_t result;
#ifdef DEBUG
        printf("\nReadying synchronization message");
        printfflush();
#endif
        // syncpkt is sent over on broadcast.
        result = call SyncSend.send(
                &spkt, sizeof(SyncMsg));
        if (result == SUCCESS)
        {
            // Everything went ok
            reportSent();
        }
        else
        {
            reportProblem();
        }
    }

    /**
```

```
      Task that readies and sends a message with the detected peaks
      Uses peakpkt to point to the payload of pkt
      totalReadings contain the readings to examine
      */
    task void sendPeakPacket()
    {
        error_t result;
        peakpkt = (PeakDetMsg*)(call PeakPacket.getPayload(&ppkt[peakClient],
                    sizeof(PeakDetMsg)));
        if (peakpkt != NULL)
        {
            peakpkt->nodeid = TOS_NODE_ID;
            peakpkt->syncid = lastSyncId;
            peakpkt->flags = 0x00;
            // Save the information from the detected peaks at the
            // packet that is goind to be sent
            detectPeaks(peakpkt->peaks);
            // pkt is sent over on broadcast.
            call PeakAMPacket.setDestination(&ppkt[peakClient],
                    AM_BROADCAST_ADDR);
            result = call PeakSend.send[peakClient](
                    &ppkt[peakClient], sizeof(PeakDetMsg));
            if (result == SUCCESS)
            {
                nextPeakClient();
                reportSent();
                // Free the synchronization receiver to listen to new packets
                atomic
                    syncLocked = FALSE;
            }
            else
                reportProblem();
        }
    }


    /*********************
      Events
      ******************/


    event void Boot.booted()
    {
        atomic{
            readingsIndex = 0;
            totalIndex = 0;
            syncLocked = FALSE;
        }

        //The entry point of the program
        // Ask for the Sensor. If it is granted, initialize the Radio
        // That way the system will not listen to synchronization
        // messages before having the sensor ready to start sampling
#ifdef DEBUG
        printf("\nRequesting sensor");
        printfflush();
#endif
        call SyncStdControl.start();
        call PeakStdControl.start();

        call SensorResource.request();
```

```
        // Ready the sync packet to use
        syncpkt = (SyncMsg*)(call SyncPacket.getPayload(&spkt,
sizeof(SyncMsg)));
        if(syncpkt != NULL)
        {
            syncpkt->flag = SYNC_FLAG;
            syncpkt->src = TOS_NODE_ID;
        }
        call SyncAMPacket.setType(&spkt, AM_SYNCMSG);
        call SyncAMPacket.setDestination(&spkt, AM_BROADCAST_ADDR);

        // It's not actually necessary to send synchronizations
        // since the receiver will not send ultrasonic pulses
        //call Timer0.startPeriodic(TIMER_PERIOD_MILLI);

    }

    /**
      Periodically send a synchronization packet
      */
    event void Timer0.fired()
    {
        post sendSyncPacket();
    }

    /*********************
      Sensor events
      ******************/

    event void SensorResource.granted()
    {
        //We have been granted the resource of the sensor.
        // The radio is initialized.
        if (call RadioControl.start() != SUCCESS)
        {
            reportProblem();
        }

    }

    async event uint16_t* ReadSensor.multipleDataReady(uint16_t *buf, uint16_t
length)
    {
        // Ready the packet so we can send it:
        memcpy(totalReadings + totalIndex, buf, 2*length);
        totalIndex += length;
        if (totalIndex == TOTAL_SAMPLES)
        {
            // Do the peak detection routine and send the packet
            totalIndex = 0;

            post sendPeakPacket();
            return NULL;
        }

        return buffer;
    }

    async event error_t ReadSensor.singleDataReady(uint16_t data)
    {
        return SUCCESS;
```

```
    }


    /***************************
      Radio controlling events
      *************************/

    event void RadioControl.startDone(error_t err)
    {
        if (err == SUCCESS)
        {
        }
        else
        {
            // If the radio does not start, try again.
            reportProblem();
            call RadioControl.start();
        }
    }

    event void RadioControl.stopDone(error_t err)
    {
#ifdef DEBUG
        printf("\n\n Radio stopDone!");
        printfflush();
#endif
    }

    /***************************
      Sync events
      *************************/
    event void SyncSend.sendDone(message_t* msg, error_t err)
    {
#ifdef DEBUG
        printf("\nSynchronization sent");
        printfflush();
#endif
        //post sendPeakPacket();
    }
    event message_t* SyncReceive.receive(message_t* msg, void* payload,
uint8_t len)
    {
        message_t* ret = msg;
        reportReceived();
#ifdef DEBUG
        printf("\nSYNC event, %u, %u", len, sizeof(SyncMsg));
        printfflush();
#endif

        if (len == sizeof(SyncMsg))
        {
            atomic
                if (!syncLocked)
                {
                    syncpkt = (SyncMsg*)payload;
#ifdef DEBUG
                    printf("\n--Synchronization received--\n");
#endif
                    if (syncpkt->flag == SYNC_FLAG)
                    {
```

```
                        lastSyncId = syncpkt->src;
                        syncLocked = TRUE;
                        readySensor();
                    }
                }
        }
        else
        {
            reportProblem();
        }
        return ret;
    }
    /**************************
      Peak events
      *************************/
    event void PeakSend.sendDone[uint8_t client](message_t* msg, error_t err)
    {
#ifdef DEBUG
        printf("\nPeak list sent");
        printfflush();
#endif
    }
    event message_t* PeakReceive.receive(message_t* msg, void* payload,
uint8_t len)
    {
        reportReceived();
        return msg;
    }


}
```

8.2.2 Receptor

8.2.2.1 UsPulse.h

```
#ifndef USRECEIVER_H
#define USRECEIVER_H

enum{
    // Timer
    TIMER_PERIOD_MILLI = 4000,
    JIFFIES = 160,
    MAX_CLIENTS = 10,
    //flags
    SYNC_FLAG = 0x01,
    //messages constants
    AM_SYNCMSG = 6,
    // Time the pulse stays high (32Khz)
    // 8 ~= 300 us
    TIMER_PULSE = 4, //32, 1ms
    // 576 ~= 18 ms
    TIMER_PREAMBLE = 100,
};

typedef struct SyncMsg {
    uint16_t flag;
    uint16_t src;
    uint16_t dest;
} SyncMsg;
```

```
#endif
```

8.2.2.2 UsPulseAppC.nc

```
#include <Timer.h>
#include "UsPulse.h"
#include "printf.h"
#include "CC2420.h"

configuration UsPulseAppC{
}
implementation{
    components UsPulseC as App;
    components MainC;
    components LedsC;
    components new TimerMilliC() as Timer0;
    components new Alarm32khz32C() as Preamble;
    components new Alarm32khz16C() as TimerPulse;

    components CC2420CsmaC;
    components RandomC as Random;

    components HplMsp430GeneralIOC as IO;

    // Radio
    components ActiveMessageC;
    // Synchronization message
    components new AMSenderC(AM_SYNCMSG) as SyncSender;
    components new AMReceiverC(AM_SYNCMSG) as SyncReceiver;


    //Timing
    components Msp430Counter32khzC as Counter;

    //components new RadioAppC(AM_SYNCMSG) as Sync;

    /** Wiring */

    App.Boot -> MainC.Boot;
    App.Timer0 -> Timer0;
    App.Preamble -> Preamble;
    App.TimerPulse -> TimerPulse;
    App.Leds -> LedsC;

    // Radio wiring
    App.RadioControl -> ActiveMessageC;
    // Sync wiring
    App.SyncSend -> SyncSender; //Only one client
    App.SyncPacket -> SyncSender;
    App.SyncReceive -> SyncReceiver;
    //App.SyncStdControl -> Sync.StdControl;
    App.SyncAMPacket -> SyncSender;
    // Output wiring
    // XXX Funcionan: DAC0,DAC1,Port62
    App.PulseOutput -> IO.DAC0;
    App.PulseTest -> IO.DAC1;

    App.RadioBackoff -> CC2420CsmaC.RadioBackoff;
    App.Random -> Random;
```

```
    App.Counter -> Counter;
}
```

8.2.2.3 UsPulseC.nc

```
#include <Timer.h>
#include "printf.h"
#include "UsPulse.h"
#include "CC2420.h"

module UsPulseC
{
    uses
    {
        interface Boot;

        interface SplitControl as RadioControl;
        // Synchronization
        interface Receive as SyncReceive;
        //AM
        interface AMSend as SyncSend;
        interface Packet as SyncPacket;
        interface AMPacket as SyncAMPacket;
        //interface StdControl as SyncStdControl;

        interface HplMsp430GeneralIO as PulseOutput;
        interface HplMsp430GeneralIO as PulseTest;

        interface Timer<TMilli> as Timer0;
        interface Alarm<T32khz, uint32_t> as Preamble;
        interface Alarm<T32khz, uint16_t> as TimerPulse;
        interface Counter<T32khz, uint16_t> as Counter;
        interface Leds;

        interface RadioBackoff;

        interface Random;

    }

}

implementation
{
    //Timers
    uint16_t timeBeforeSync;
    uint16_t timeAfterSync;
    uint16_t timePulseLaunched;

    // Backoff time for CSMA
    uint32_t backoffTime;
    //uint32_t congestionBackoffTime;

    // Sync packet
    message_t spkt;
    SyncMsg *syncpkt;

    void reportProblem()
    {
        call Leds.led0Toggle();
```

```
    }
    void reportSent()
    {
        call Leds.led1Toggle();
    }
    void reportReceived()
    {
        call Leds.led2Toggle();
    }

    /**
      Task to send a synchronization message so other motes wait for the US
      pulse.
      Uses syncpkt* to point to the payload of syncpkt
      */
    void sendSyncPacket()
    {
        error_t result;
        // syncpkt is sent over on broadcast.
        result = call SyncSend.send(AM_BROADCAST_ADDR,
                &spkt, sizeof(SyncMsg));
        if (result == SUCCESS)
        {
            //TIME
            call PulseTest.set();
            timeAfterSync = call Counter.get();
            // Everything went ok
            reportSent();
        }
        else
        {
            reportProblem();
        }
    }


    /*********************
      Events
      ******************/


    event void Boot.booted()
    {
        //The entry point of the program
        //call SyncStdControl.start();

        // Initialize the sync packet, as it wil be constant throughout time
        syncpkt = (SyncMsg*)(call SyncPacket.getPayload(&spkt,
sizeof(SyncMsg)));
        if(syncpkt != NULL)
        {
            syncpkt->flag = SYNC_FLAG;
            syncpkt->src = TOS_NODE_ID;
        }
        call SyncAMPacket.setType(&spkt, AM_SYNCMSG);
        // END

        // Initialize the output pin
        call PulseOutput.selectIOFunc();
        call PulseOutput.makeOutput();
        // Set the pin to LOW
```

```
        call PulseOutput.clr();
        // Initialize the output pin
        call PulseTest.selectIOFunc();
        call PulseTest.makeOutput();
        // Set the pin to LOW
        call PulseTest.clr();

        call RadioControl.start();


    }
    /**
      CMSA events

      They use the same formula as vanilla CSMA access from TinyOS 2.x
      */
    async event void RadioBackoff.requestInitialBackoff(message_t * ONE msg)
    {
        backoffTime = call Random.rand16()
            % (0x1F * CC2420_BACKOFF_PERIOD) + CC2420_MIN_BACKOFF;
        call RadioBackoff.setInitialBackoff (backoffTime);

    }

    async event void RadioBackoff.requestCongestionBackoff(message_t * ONE
msg)
    {
        backoffTime = call Random.rand16()
            % (0x7 * CC2420_BACKOFF_PERIOD) + CC2420_MIN_BACKOFF;
        call RadioBackoff.setCongestionBackoff( backoffTime );
    }

    async event void RadioBackoff.requestCca(message_t * ONE msg)
    {
    }

    /**
      Periodically send a synchronization packet
      */
    event void Timer0.fired()
    {

        //TIME
        timeBeforeSync = call Counter.get();
        sendSyncPacket();
        //TIME
        call PulseTest.clr();
        //call PulseOutput.toggle();
        // We use the same backoff the radio is using
        // in order to eliminate the random delay when sending the US pulse.
        call Preamble.start(TIMER_PREAMBLE + backoffTime);
    }

    async event void Preamble.fired()
    {
        timePulseLaunched = call Counter.get();
        call PulseOutput.set();
        call TimerPulse.start(TIMER_PULSE);
        reportReceived();
    }
```

```
    /**
      Return the pin to LOW
      */
    async event void TimerPulse.fired()
    {
        call PulseOutput.clr();
        reportReceived();
        printf("Timing:\n");
        printf(" timeBeforeSync = %u, %d\n",
                timeBeforeSync, timeAfterSync-timeBeforeSync);
        printf(" timeAfterSync = %u, %d\n",
                timeAfterSync, timePulseLaunched-timeAfterSync);
        printf(" timePulseLaunched = %u\n", timePulseLaunched);
        printfflush();
    }

    async event void Counter.overflow()
    {
    }

    /*********************
      Pulse events
      ******************/




    /***************************
      Radio controlling events
      *************************/

    event void RadioControl.startDone(error_t err)
    {
        if (err == SUCCESS)
        {
            // If the radio is granted, start the timer
            call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
        }
        else
        {
            // If the radio does not start, try again.
            reportProblem();
            call RadioControl.start();
        }
    }

    event void RadioControl.stopDone(error_t err)
    {
#ifdef DEBUG
        printf("\n\n Radio stopDone!");
        printfflush();
#endif
    }

    /***************************
      Sync events
      *************************/
    event void SyncSend.sendDone(message_t* msg, error_t err)
    {
#ifdef DEBUG
        printf("\nSynchronization sent");
```

```
        printfflush();
#endif
    }

    event message_t* SyncReceive.receive(message_t* msg, void* payload,
uint8_t len)
    {
        reportReceived();
        return msg;
    }


}
```

8.3 Emisor y receptor unidos

8.3.1 UsReceiver.h

```
#ifndef USRECEIVER_H
#define USRECEIVER_H

enum{
    // Time between synchronizations
    TIMER_PERIOD_MILLI = 5000,
    // Conversor
    VOLTAGE_CONVERSION = (3/4096),
    // Sampling
    NUM_SAMPLES = 16,
    //MSG_SAMPLES = 96,
    TOTAL_SAMPLES = 224, //480 en 160us, 224 si printf, //576, %16
    // Since it is saved in a uint16_t, a value is needed for every 16
samples.
    // Save a bit for every measure, 1 meaning there is a peak and 0 meaning
    // there isn't.
    NUM_PEAKS = (TOTAL_SAMPLES/16),
    //560 en 140 us
    //MSGS_TO_SEND = (TOTAL_SAMPLES/MSG_SAMPLES), //Must be integer
    JIFFIES = 160, //160, 140 funciona
    SAMPLES_TO_TIME = (JIFFIES), //Find real formula
    SOUND_SPEED = 343, //m/s
    NO_PEAK = -1, //0
    // Times the peaks will be averaged. Must be less than 255.
    NUM_AVERAGES = 3,
    // Number of possible peaks that will be considered in the averaging
    NUM_POSSIBLE_PEAKS = 60,
    // Maximum value that will be considered the same peak, in samples
    // For instance, 3 => up to x-3 and x+3 will be considered the same peak
    // when doing the channel model.
    PEAK_MARGIN = 1,
    // Clock value that will be considered valid
#ifdef USE_SRF02
    VALID_CLOCK_OFFSET = 84,
    TIMER_SRF = 100,
    SRF_PREAMBLE = 191,
#else
    VALID_CLOCK_OFFSET = 8,
#endif
    // Threshold that will be considered noise.
    // To calculate a new threshold, divide desired voltage by
    // VOLTAGE_CONVERSION. For instance, for a threshold of 0.5 V,
```

```
    // this constant must be 0.5*4096/3 = 682.6666 => 683
    NOISE_THRESHOLD = 100,
    //1638 -> 1.2V
    //1229->0.9V //273, //273->0.2V //*****
    //540 ~= 0.4V
    // for the peak detection algorithm
    TRACK_MAX = 1,
    TRACK_MIN = 0,
    // Minimum number of samples between succesful peak detections
    // If PEAK_MARGIN SAMPLES == 0, leave no margin
    PEAK_MARGIN_SAMPLES = 0, //*****
    /*(unused) XXX

    // Queues
    UART_QUEUE_LEN = 6,
    RADIO_QUEUE_LEN = 6,
    MAX_CLIENTS = 10,
    // Flags
    IS_MODEL = 0x01,
    LAST_MSG = 0x02,
    */
    // Sync packet flags
    SYNC_FLAG = 0x01,
    GET_MODEL_FLAG = 0x02,
    ALIVE_FLAG = 0x04,
    //messages constants
    AM_USRECEIVERMSG = 4,
    AM_SYNCMSG = 6,
    AM_PEAKDETMSG = 5,
    // Time the pulse stays high (32Khz)
    // 8 ~= 300 us
    TIMER_PULSE = 8, //6 //con 10 funciona
    // 576 ~= 18 ms
    TIMER_PREAMBLE = 75, //74.4, 2525 us of delay
    // Local sleep interval, in milliseconds
    LPL_LOCAL_SLEEP = 1000,
    // Time until new synchronization packets are accepted (in ms)
    TIMER_ACCEPT_SYNC = 1000,
    // Number of bits in an array position
    NUM_BITS = 16,
    // XXX Timer para enviar el paquete de picos (temporal)
    TIMER_PEAK = 100,

};
// If this time passes after the last message sent, send a message to
// confirm the node is still alive
#define TIMER_STILL_ALIVE 30000

// To work with bits
#define BITMASK(b) (1 << ((b) % NUM_BITS))
#define BITSLOT(b) ((b) / NUM_BITS)
#define BITSET(a, b) ((a)[BITSLOT(b)] |= BITMASK(b))
#define BITCLEAR(a, b) ((a)[BITSLOT(b)] &= ~BITMASK(b))
#define BITTEST(a, b) ((a)[BITSLOT(b)] & BITMASK(b))
#define BITNSLOTS(nb) ((nb + NUM_BITS - 1) / NUM_BITS)

typedef struct SyncMsg {
    uint16_t flag;
    uint16_t src;
    uint16_t dest;
} SyncMsg;
```

```
typedef struct PeakDetMsg {
    uint16_t nodeid;
    uint16_t syncid;
    uint16_t flags;
    uint16_t peaks[NUM_PEAKS];
} PeakDetMsg;


#endif
```

8.3.2 UsReceiverAppC.nc

```
// Configuration file for UsReceiver.

#include <Timer.h>
#include "UsReceiver.h"
#include "printf.h"

configuration UsReceiverAppC{
}
implementation{
    components UsReceiverC as App;
    components MainC;
    components LedsC;
    components new TimerMilliC() as Timer0;
    components new TimerMilliC() as TimerSync;
    components new TimerMilliC() as TimerStillAlive;
    components new TimerMilliC() as TimerPeak;

    // Analog channels
    components new Msp430Adc12ClientC() as AdcClient;

    // Radio
    components ActiveMessageC;
    // Information message
    components new AMSenderC(AM_PEAKDETMSG) as PeakSender;
    components new AMReceiverC(AM_PEAKDETMSG) as PeakReceiver;
    // Synchronization message
    components new AMSenderC(AM_SYNCMSG) as SyncSender;
    components new AMReceiverC(AM_SYNCMSG) as SyncReceiver;

    //components new RadioAppC(AM_USRECEIVERMSG) as Radio;
    //components new RadioAppC(AM_SYNCMSG) as Sync;
    //components new RadioAppC(AM_PEAKDETMSG) as Peak;

    components new Alarm32khz32C() as Preamble;
#ifdef USE_SRF02
    components SRF02C as SRF02;
    components new TimerMilliC() as TimerSRF;
#else
    // Pulse transmission
    components new Alarm32khz16C() as TimerPulse;

    components HplMsp430GeneralIOC as IO;
#endif
    //Timing
    components Msp430Counter32khzC as Counter;

    // Low Power Listening
```

```
#ifdef LOW_POWER_LISTENING
    components CC2420ActiveMessageC as LplRadio;
#endif

    /** Wiring */

    App.Boot -> MainC.Boot;
    App.Timer0 -> Timer0;
    App.TimerSync -> TimerSync;
    App.TimerStillAlive -> TimerStillAlive;
    App.TimerPeak -> TimerPeak;
    App.Leds -> LedsC;

    // Sensor wiring
    App.SensorResource -> AdcClient.Resource;
    App.ReadSensor -> AdcClient.Msp430Adc12SingleChannel;

    // Radio wiring
    App.RadioControl -> ActiveMessageC;
    // Sync wiring
    App.SyncSend -> SyncSender; //Only one client
    App.SyncPacket -> SyncSender;
    App.SyncReceive -> SyncReceiver;
    //App.SyncStdControl -> Sync.StdControl;
    App.SyncAMPacket -> ActiveMessageC;
    // Peak detection wiring
    App.PeakSend -> PeakSender;
    App.PeakPacket -> PeakSender;
    App.PeakReceive -> PeakReceiver;
    //App.PeakStdControl -> Peak.StdControl;
    App.PeakAMPacket -> ActiveMessageC;

    App.Preamble -> Preamble;

#ifdef USE_SRF02
    App.SRF02 -> SRF02;
    App.TimerSRF -> TimerSRF;
#else
    // Pulse
    App.PulseOutput -> IO.Port62; //IO.DAC0;//
    App.TimerPulse -> TimerPulse;
#endif

    App.Counter -> Counter;

#ifdef LOW_POWER_LISTENING
    App.LowPowerListening -> LplRadio;
#endif
}
```

8.3.3 UsReceiverC.nc

```
#include <Timer.h>
#include "printf.h"
#include "UsReceiver.h"
#include "SRF02.h"

module UsReceiverC
{
    uses
    {
```

```
        interface Boot;

        interface SplitControl as RadioControl;
        // Synchronization
        interface Receive as SyncReceive;
        interface AMSend as SyncSend;
        interface Packet as SyncPacket;
        interface AMPacket as SyncAMPacket;
        //interface StdControl as SyncStdControl;
        // Peaks
        interface Receive as PeakReceive;
        interface AMSend as PeakSend;
        interface Packet as PeakPacket;
        interface AMPacket as PeakAMPacket;
        //interface StdControl as PeakStdControl;

        interface Msp430Adc12SingleChannel as ReadSensor;
        interface Resource as SensorResource;

        interface Timer<TMilli> as Timer0;
        interface Timer<TMilli> as TimerSync;
        interface Timer<TMilli> as TimerStillAlive;
        // XXX
        interface Timer<TMilli> as TimerPeak;
        interface Leds;

        interface Alarm<T32khz, uint32_t> as Preamble;
#ifdef USE_SRF02
        interface SRF02;
        interface Timer<TMilli> as TimerSRF;
#else
        interface HplMsp430GeneralIO as PulseOutput;

        interface Alarm<T32khz, uint16_t> as TimerPulse;
#endif
        interface Counter<T32khz, uint16_t> as Counter;


#ifdef LOW_POWER_LISTENING
        interface LowPowerListening;
#endif

    }

}

implementation
{
    const msp430adc12_channel_config_t config = {
        INPUT_CHANNEL_A0,           //input channel
        REFERENCE_AVcc_AVss,        //reference voltage
        REFVOLT_LEVEL_NONE,         //reference voltage level
        SHT_SOURCE_ADC12OSC,        //clock source sample-hold-time
        SHT_CLOCK_DIV_1,            //clock divider sample-hold-time
        SAMPLE_HOLD_384_CYCLES,     //sample-hold-time
        SAMPCON_SOURCE_SMCLK,       //clock source sampcon signal
        SAMPCON_CLOCK_DIV_1         //clock divider sampcon
    };

    // Buffer to save temporary measures.
    uint16_t buffer[NUM_SAMPLES];
```

```
// Last measures captured
// XXX uint16_t lastReadings[MSG_SAMPLES];
uint16_t totalReadings[TOTAL_SAMPLES];
uint16_t totalIndex;

// Averaged readings
uint16_t averagedReadings[TOTAL_SAMPLES];
uint8_t timesAveraged;

// Peaks of the environment model
uint16_t modelPeaks[NUM_PEAKS];
bool isModel;

// Last peaks saved
uint16_t lastPeaks[NUM_PEAKS];
// XXX uint8_t readingsIndex;

// NodeID of the last synchronization received
uint16_t lastSyncId;

// To count time elapsed.
bool firstTime;
uint16_t timePulse;
uint16_t timeReading;

// Sync packet
message_t spkt;
// If sync is locked, do not take new measures
bool syncLocked;
// XXX uint8_t peakClient;
SyncMsg *syncpkt;

// Message containing the peak detection results to be sent on radio.
PeakDetMsg *peakpkt;
message_t ppkt;// XXX [MAX_CLIENTS];

// Source of the synchronization packets
am_addr_t syncSource;

// XXX
error_t srfResult;

// TEST SAMPLES
/*
   XXX
uint16_t testReadings[TOTAL_SAMPLES] =
{
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 500, 550, 600, 300, 250, 400, 1000, 800, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    // 10
    0, 10, 20, 30, 10, 40, 10, 1000, 1550, 1540, 1560, 700, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    // 20
    0, 1000, 1100, 1000, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    // 30
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 200, 100, 200, 100, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 100, 200, 4000, 2000, 600, 3000, 3000, 0, 0, 0, 0, 0, 0, 0, 0,
    // 36
}
    ;
    */
// END TEST SAMPLES
task void sendPeakPacket();

void reportProblem()
{
    call Leds.led0Toggle();
}
void reportSent()
{
    call Leds.led1Toggle();
}
void reportReceived()
{
    call Leds.led2Toggle();
}

/**
  XXX Sets next peakClient for the new use XXX
void nextPeakClient()
{
    peakClient++;
    if (peakClient >= MAX_CLIENTS)
       peakClient = 0;
}
  */


/**
  XXX Prints peaks locations on the terminal XXX
  */
/*
void printPeaks()
```

```
    {
#ifdef PRINT_PEAKS
        uint8_t i;
        printf("Peaks detected:\n");
        for (i = 0; i < NUM_PEAKS; i++)
        {
            if (lastPeaks[i] != NO_PEAK)
            {
                printf("peak %u = %u, time = %u us, distance = %llu\n", i,
                        lastPeaks[i],
                        lastPeaks[i]*SAMPLES_TO_TIME,
                        lastPeaks[i]*SAMPLES_TO_TIME*SOUND_SPEED);
            }
        }
        printfflush();
#endif
        printf("Time elapsed: %u - %u = %d\n", timePulse,
                timeReading, timePulse-timeReading);
        printfflush();
    }
*/

    /**
      Compares peaks with model peaks, eliminating peaks caused
      by the channel model.
      */
    task void compareModel()
    {
        uint8_t i;
        bool notZero = FALSE;

#ifdef PRINT_PEAKS
        printf("Peaks: \n");
        for (i = 0; i < NUM_PEAKS; i++)
        {
            printf("%X ", lastPeaks[i]);
        }
        printf("\n");
        printfflush();
#endif
        for (i = 0; i < NUM_PEAKS; i++)
        {
            // modelPeaks is negated to transform 1s into 0s
            lastPeaks[i] &= ~modelPeaks[i];
            // Check if there is a peak
            if (lastPeaks[i] != 0)
                notZero = TRUE;
        }
#ifdef PRINT_PEAKS
        printf("Peaks w/o model: \n");
        for (i = 0; i < NUM_PEAKS; i++)
        {
            printf("%X ", lastPeaks[i]);
        }
        printf("\n");
        printfflush();
#endif
        // As long as *one* peak is different from the model, send
        // the packet
        if (notZero)
        {
```

```
            // XXX
#ifdef USE_SRF02
            call TimerPeak.startOneShot(TIMER_PEAK);
#else
            post sendPeakPacket();
#endif
        }
    }

    /**
      Sets the model into an usable mode: 1s where model peaks are
      and 0s in the rest of the array, so a simple '&' of the new array
      and the negated model will tell whether there are new peaks or not.
      */
    task void setModel()
    {
        uint16_t i;
        uint8_t j;
#ifdef PRINT_PEAKS
        printf("Model: \n");
        for (i = 0; i < NUM_PEAKS; i++)
        {
            printf("%X ", modelPeaks[i]);
        }
        printf("\n");
        printfflush();
#endif
        for (i = 0; i < TOTAL_SAMPLES; i++)
        {
            // Leave a PEAK_MARGIN margin before and after the model peak.
            if (BITTEST(modelPeaks, i) != 0)
            {
                for (j = 1; j <= PEAK_MARGIN; j++)
                {
                    // Check if the position is valid before putting it, in
                    // order to avoid corrupting data.
                    if ((i+j) < TOTAL_SAMPLES)
                        BITSET(modelPeaks, (i+j));
                    if ((i-j) >= 0)
                        BITSET(modelPeaks, (i-j));
                }
                // Set i one above the PEAK_MARGIN to avoid triggering
                // the 'if' with the newly put value(s).
                i += PEAK_MARGIN;
            }
        }
#ifdef PRINT_PEAKS
        printf("New Model: \n");
        for (i = 0; i < NUM_PEAKS; i++)
        {
            printf("%X ", modelPeaks[i]);
        }
        printf("\n");
        printfflush();
#endif

    }


    /**
      Set averaged readings to 0 again.
```

```
     */
    void resetReadings()
    {
        // uint16_t, so it has 2 bytes per sample.
        memset(averagedReadings, 0, 2*TOTAL_SAMPLES);
    }

    /**
      Simulates a PREG register, a one edge register with enable,
      in order to detect peaks

      This tracks the maximum value entered when enabled, and the minimum
      value entered when disabled.
      */
    int preg(bool enable, int in, int lastPreg)
    {
        if (enable)
        {
            if (in > lastPreg)
            {
                return in;
            }
            else
                return lastPreg;
        }
        else
        {
            if (in <= lastPreg)
            {
                return in;
            }
            else
                return lastPreg;
        }
    }


    /**
      Detects peaks on a sample buffer and saves their coordinates on
      lastPeaks.
      */
    task void detectPeaks()
    {
        uint16_t i;
#ifdef TEST_PEAKS
        // XXX
        for (i = 0; i < NUM_PEAKS; i++)
        {
            if (i == 0)
                lastPeaks[i] = 185;
            else
                lastPeaks[i] = NO_PEAK;
        }
#else
        // XXX uint8_t peakIndex = 0;
        int noise = NOISE_THRESHOLD;
        int pr = 0;
        bool enablePreg;
        int partial = 0;
        // The jump from TRACK_MAX to TRACK_MIN signifies a detected peak
        uint8_t lastOut = TRACK_MAX;
```

```
        uint8_t out = TRACK_MAX;
        uint8_t sign = 0;
        // Initialize margin so first samples are detectable
        uint8_t margin = PEAK_MARGIN_SAMPLES;

        // Reset lastPeaks
        if (isModel)
            memset(modelPeaks, 0, sizeof(modelPeaks));
        else
            memset(lastPeaks, 0, sizeof(lastPeaks));

        for (i = 0; i < TOTAL_SAMPLES; i++)
        {
            lastOut = out;

            if (lastOut == TRACK_MIN)
                noise = NOISE_THRESHOLD;
            else
                noise = -NOISE_THRESHOLD;


            partial = averagedReadings[i] - pr;
            // XXX
            //printf("%u: (%d)\n", i, lastOut);
            //printf("%d ", partial);

            if (partial >= 0)
                sign = 0;
            else if (partial < 0)
                sign = 1;

            enablePreg = lastOut^sign;
            pr = preg(enablePreg, averagedReadings[i],pr);

            if (partial > noise)
                out = TRACK_MAX;
            else
                out = TRACK_MIN;

            /*
            If PEAK_MARGIN_SAMPLES samples have passed, enable detection again
            If PEAK_MARGIN SAMPLES == 0, leave no margin
            */
            if (margin < PEAK_MARGIN_SAMPLES)
            {
                margin++;
            }
            else if ((out == TRACK_MIN)&&(lastOut == TRACK_MAX))
            {
                // XXX NOT INVERSE PEAK DETECTION XXX
                // Peak detected
#ifdef PRINT_PEAKS
                printf("Peak detected: %u, %u, sample %u\n",
                        averagedReadings[i-1], averagedReadings[i], i);
                printfflush();
#endif
                // Save the peak information
                // First samples are saved on the rightmost part of the
                // array
                if (isModel)
                    BITSET(modelPeaks, i);
```

```
                else
                    BITSET(lastPeaks, i);

                /*
                   XXX
                peakIndex++;
                // Once the first NUM_PEAKS are detected, exit the function
                if (peakIndex >= NUM_PEAKS)
                {
                    printPeaks(lastPeaks);
                    return;
                }
                */
            }
        }
        // XXX printPeaks();
        // Averaged readings must be reset to get the new values.
        resetReadings();
        if (isModel)
        {
            isModel = FALSE;
            post setModel();
        }
        else
            post compareModel();
#endif
    }


    /**
      Readies the sensor and calls getData() to start getting samples
     */
    void readySensor()
    {
        error_t result;
        error_t request;
        //First we must initialize the sensor.
        //Since it is MultipleRepeat, we must initialize it each time.
#ifdef USE_SRF02
        request = call SensorResource.request();
        // XXX
#ifdef DEBUG
        printf("request = %d\n", request);
#endif
#else
        request = SUCCESS;
#endif
        result = call ReadSensor.configureMultipleRepeat(&config,
                buffer,
                NUM_SAMPLES,
                JIFFIES);
        if ((result == SUCCESS)&&(request == SUCCESS))
        {
            //If the initalization was correct, start reading data.
#ifdef DEBUG
            printf("Calling getData\n");
            printfflush();
#endif
#ifdef USE_SRF02
            srfResult = call SRF02.setCommand(SEND_ULTRA);//REQ_MEAS_CM);
            call Preamble.start(SRF_PREAMBLE);
```

```
#else
            call ReadSensor.getData();
            call Preamble.start(TIMER_PREAMBLE);
#endif
        }
        else
        {
#ifdef DEBUG
            printf("ERROR: configuring ReadSensor\n");
            printfflush();
#endif
            reportProblem();
            //XXX
#ifdef USE_SRF02
            call TimerSRF.startOneShot(TIMER_SRF);
#else
            readySensor();
#endif
        }
    }


    /**
      Task to send a synchronization message so other motes wait for the US
      pulse.
      Uses syncpkt* to point to the payload of syncpkt
     */
    task void sendSyncPacket()
    {
        error_t result;
#ifdef DEBUG
        printf("\nReadying synchronization message\n");
        printfflush();
#endif
        // syncpkt is sent to the mote that sent the message asking for
        // measures.
        call SyncAMPacket.setDestination(&spkt, syncSource);
        syncpkt = (SyncMsg*)(call SyncPacket.getPayload(&spkt,
                    sizeof(SyncMsg)));
        if(syncpkt != NULL)
        {
            syncpkt->dest = syncSource;
        }
        result = call SyncSend.send(syncSource,
                &spkt, sizeof(SyncMsg));
        if (result == SUCCESS)
        {
            // Everything went ok

            // Reset flags
            syncpkt->flag = SYNC_FLAG;
            reportSent();
        }
        else
        {
#ifdef DEBUG
            printf("SYNC send ERROR: %u\n", result);
            printfflush();
#endif
            reportProblem();
        }
```

```
    }

    /**
      Averages the samples the NUM_AVERAGES times
     */
    task void averageReadings()
    {
        uint16_t i;
        uint16_t offset = timeReading-timePulse;
#ifdef USE_SRF02
#ifdef DEBUG
        // XXX
        printf("SRF02 result = %d\n", srfResult);
        printfflush();
#endif
#endif
        // Only average if the clock skew is a valid value, to avoid using
        // displaced samples.
        // XXX
        if ((offset <= VALID_CLOCK_OFFSET + 1) &&
                (offset >= VALID_CLOCK_OFFSET - 1) &&
                (srfResult == SUCCESS))
        {
            for (i = 0; i < TOTAL_SAMPLES; i++)
            {
                averagedReadings[i] += (totalReadings[i]/NUM_AVERAGES);
            }

            timesAveraged++;
            printf("timesAveraged = %d\n", timesAveraged);
            if (timesAveraged >= NUM_AVERAGES)
            {
                // Send the packet
                timesAveraged = 0;
                post detectPeaks();
                // XXX post sendPeakPacket();
            }
            else
            {
                // If it's not the last time, continue:
#ifdef USE_SRF02
                call TimerSRF.startOneShot(TIMER_SRF);
#else
                readySensor();
#endif
            }
        }
        else
        {
            printf("OFFSET ERROR: %d\n", offset);
            // Incorrect, so do it again. This will repeat until the clock
            // value is valid.
#ifdef USE_SRF02
            call TimerSRF.startOneShot(TIMER_SRF);
#else
            readySensor();
#endif
        }

    }
```

```
    /**
      Task that readies and sends a message with the detected peaks
      Uses peakpkt to point to the payload of pkt
      totalReadings contain the readings to examine
     */
    task void sendPeakPacket()
    {
        error_t result;
        peakpkt = (PeakDetMsg*)(call
PeakPacket.getPayload(&ppkt/*[peakClient]*/,
                    sizeof(PeakDetMsg)));
        if (peakpkt != NULL)
        {
            peakpkt->nodeid = TOS_NODE_ID;
            peakpkt->syncid = lastSyncId;
            peakpkt->flags = 0x00;
            // Save the information from the detected peaks at the
            // packet that is goind to be sent
            memcpy(peakpkt->peaks, lastPeaks, sizeof(lastPeaks));
            // XXX detectPeaks(peakpkt->peaks);
            // XXX call PeakAMPacket.setGroup(&ppkt/*[peakClient]*/,
AM_GROUP);

            // pkt is sent to the node that requested the measure.
            call PeakAMPacket.setDestination(&ppkt/*[peakClient]*/,
                    syncSource);
            // XXX AM_BROADCAST_ADDR);
            result = call PeakSend.send(syncSource,
                    &ppkt/*[peakClient]*/, sizeof(PeakDetMsg));
            if (result == SUCCESS)
            {
                //nextPeakClient();
                reportSent();
#ifdef DEBUG
                // XXX Necesario? XXX
                printf("Peaks sent\n");
                printfflush();
#endif
                // Wait and free the synchronization receiver to listen
                // to new packets
                call Timer0.startOneShot(TIMER_ACCEPT_SYNC);
            }
            else
            {
#ifdef DEBUG
                printf("PEAK send ERROR: %u\n", result);
                printfflush();
#endif
                reportProblem();
            }
        }
    }


    /*********************
      Events
      ******************/


    event void Boot.booted()
    {
```

```
        atomic{
            // TODO: Repasar cuales de estos hacen falta
            //readingsIndex = 0;
            totalIndex = 0;
            timesAveraged = 0;
            syncLocked = FALSE;
            firstTime = TRUE;
            isModel = TRUE;
        }
        //The entry point of the program

#ifdef DEBUG
        printf("Requesting sensor\n");
        printfflush();
#endif
        // XXX
        //call SyncStdControl.start();
        //call PeakStdControl.start();

        // Ask for the Sensor. If it is granted, initialize the Radio
        // That way the system will not listen to synchronization
        // messages before having the sensor ready to start sampling

        // FIXME: Sin SRF02, es necesario seguro.
        //call SensorResource.request();
#ifdef USE_SRF02
        call SRF02.getFirmware();
#else
        srfResult = SUCCESS;
        call SensorResource.request();
        call RadioControl.start();
        // Initialize the output pin
        call PulseOutput.selectIOFunc();
        call PulseOutput.makeOutput();
        // Set the pin to LOW
        call PulseOutput.clr();
#endif

        // Ready the base of the synchronization packet to use, since
        // it is going to be more or less constant during the life of the mote
        syncpkt = (SyncMsg*)(call SyncPacket.getPayload(&spkt,
                    sizeof(SyncMsg)));
        if(syncpkt != NULL)
        {
            syncpkt->flag = SYNC_FLAG;
            syncpkt->src = TOS_NODE_ID;
            syncpkt->dest = AM_BROADCAST_ADDR;
        }
        call SyncAMPacket.setType(&spkt, AM_SYNCMSG);
        call SyncAMPacket.setDestination(&spkt,
                AM_BROADCAST_ADDR);

#ifdef LOW_POWER_LISTENING
        // Enable Low Power Listening
        call LowPowerListening.setLocalSleepInterval(LPL_LOCAL_SLEEP);
#endif

        // Start the "Still alive" timer
        call TimerStillAlive.startPeriodic((uint32_t)TIMER_STILL_ALIVE);

#ifdef PERIODIC_SAMPLING
```

```
        // It's not actually necessary to send synchronizations
        // since the receiver will not send ultrasonic pulses
        //call TimerSync.startPeriodic(TIMER_PERIOD_MILLI);
        call TimerSync.startOneShot(TIMER_PERIOD_MILLI);
#endif
    }

    /*********************
      Timer events
      ******************/

    /*
       XXX XXX XXX
       */
    event void TimerPeak.fired()
    {
        post sendPeakPacket();
    }

    /**
      Wait to unlock the sync reception
      */
    event void Timer0.fired()
    {
        // XXX post sendSyncPacket();
        atomic
            syncLocked = FALSE;
    }

    /**
      Periodically start the measure and averaging mode.
     */
    event void TimerSync.fired()
    {
#ifdef PERIODIC_SAMPLING
        lastSyncId = 0;
        syncLocked = TRUE;
        syncSource = 0;
#endif
        readySensor();
#ifdef PERIODIC_SAMPLING
        call TimerSync.startOneShot(TIMER_PERIOD_MILLI);
#endif
        // XXX post sendSyncPacket();
    }

    /**
      Communicate with other nodes so they know the node is
      still correctly working.
      */
    event void TimerStillAlive.fired()
    {
#ifdef DEBUG
        printf("I'm not dead!\n");
        printfflush();
#endif
        // Mark message as an Alive control message
        syncpkt->flag |= ALIVE_FLAG;
        post sendSyncPacket();
    }
```

```
    /**
      Wait a time until sending the pulse
      */
    async event void Preamble.fired()
    {
#ifdef USE_SRF02
        // XXX
        call ReadSensor.getData();
        //srfResult = call SRF02.setCommand(SEND_ULTRA);//REQ_MEAS_CM);
#else
        call PulseOutput.set();
        call TimerPulse.start(TIMER_PULSE);
#endif
        timePulse = call Counter.get();
    }

#ifdef USE_SRF02
    // TODO: Placeholder for strictly SRF02-related timers.
    event void TimerSRF.fired()
    {
        readySensor();
        //call SRF02.getEcho();
    }
#else
    /**
      Return the pin to LOW
      */
    async event void TimerPulse.fired()
    {
        call PulseOutput.clr();
    }
#endif

    async event void Counter.overflow()
    {
    }

#ifdef USE_SRF02
    /*********************
      SRF02 events
      ******************/
    async event void SRF02.setCommandDone( error_t error )
    {
        printf("setCommandDone\n");
        printfflush();
        //call TimerSRF.startOneShot(TIMER_SRF);
        return;
    }
    async event void SRF02.getFirmwareDone(error_t error, uint8_t val)
    {
#ifdef DEBUG
        printf("Got firmware %d\n", val);
        printfflush();
#endif
        call SyncSend.send(AM_BROADCAST_ADDR,
                &spkt, sizeof(SyncMsg));
        return;
    }
    async event void SRF02.getEchoDone(error_t error, uint16_t val)
    {
#ifdef DEBUG
```

```
        printf("Echo value: %d\n", val);
        printfflush();
#endif
        return;
    }
    async event void SRF02.getMinDistanceDone(error_t error, uint16_t val)
    {
        return;
    }
#endif

    /*********************
      Sensor events
      ******************/

    event void SensorResource.granted()
    {
        // We have been granted the resource of the sensor.
        // The radio is initialized.
        /*
        if (call RadioControl.start() != SUCCESS)
        {
            reportProblem();
        }
        */

    }

    async event uint16_t* ReadSensor.multipleDataReady(uint16_t *buf, uint16_t
length)
    {
        // Ready the packet so we can send it:
        memcpy(totalReadings + totalIndex, buf, 2*length);
        if (firstTime)
        {
            timeReading = call Counter.get();
            firstTime = FALSE;
        }
        totalIndex += length;
        if (totalIndex == TOTAL_SAMPLES)
        {
            // Do the peak detection routine and send the packet
            totalIndex = 0;
            firstTime = TRUE;
            // Average the received readings.
            post averageReadings();
            // Do not return any buffer, so the data adquisition stops.
#ifdef USE_SRF02
            call SensorResource.release();
#endif
            return NULL;
        }

        // Return the used buffer to continue capturing data.
        return buffer;
    }

    async event error_t ReadSensor.singleDataReady(uint16_t data)
    {
        return SUCCESS;
    }
```

```
    /***************************
      Radio controlling events
      *************************/

    event void RadioControl.startDone(error_t err)
    {
        if (err == SUCCESS)
        {
#ifdef DEBUG
            printf("\nRadio startDone!\n");
            printfflush();
#endif
        }
        else
        {
#ifdef DEBUG
            printf("\nERROR: Radio start!\n");
            printfflush();
#endif
            // If the radio does not start, try again.
            reportProblem();
            call RadioControl.start();
        }
    }

    event void RadioControl.stopDone(error_t err)
    {
#ifdef DEBUG
        printf("\nRadio stopDone!\n");
        printfflush();
#endif
    }

    /***************************
      Sync events
      *************************/

    event void SyncSend.sendDone(message_t* msg, error_t err)
    {
#ifdef DEBUG
        printf("Synchronization sent\n");
        printfflush();
#endif
        //post sendPeakPacket();
    }
    event message_t* SyncReceive.receive(message_t* msg, void* payload,
uint8_t len)
    {
        SyncMsg *recSyncpkt;
        message_t* ret = msg;
        reportReceived();

        if (len == sizeof(SyncMsg))
        {
            atomic
                if (!syncLocked)
                {
                    recSyncpkt = (SyncMsg*)payload;
```

```
#ifdef DEBUG
                    printf("\n--Synchronization received--\n");
#endif
                    // If the packet is for this node or for
                    //everyone, continue
                    if (((recSyncpkt->flag & SYNC_FLAG) == SYNC_FLAG) &&
                            ((recSyncpkt->dest == TOS_NODE_ID) ||
                             (recSyncpkt->dest == AM_BROADCAST_ADDR)))
                    {
                        if ((recSyncpkt->flag & GET_MODEL_FLAG) ==
GET_MODEL_FLAG)
                            isModel = TRUE;
                        lastSyncId = recSyncpkt->src;
                        syncLocked = TRUE;
                        readySensor();
                        syncSource = call SyncAMPacket.source(msg);
                        post sendSyncPacket();
                    }
                }
        }
        else
        {
#ifdef DEBUG
            printf("ERROR: SyncReceive!\n");
            printfflush();
#endif
            reportProblem();
        }
        return ret;
    }

    /*****************************
      Peak events
      *************************/
    event void PeakSend.sendDone(message_t* msg, error_t err)
    {
        // restart still alive timer.
        call TimerStillAlive.stop();
        call TimerStillAlive.startPeriodic((uint32_t)TIMER_STILL_ALIVE);
#ifdef DEBUG
        printf("Peak list sent\n");
        printf("error_t = %u\n", err);
        printfflush();
#endif
    }
    event message_t* PeakReceive.receive(message_t* msg, void* payload,
uint8_t len)
    {
        reportReceived();
        return msg;
    }

}
```

8.3.4 SRF02.h

```
#ifndef SRF02_H
#define SRF02_H
enum {
```

```
    REQ_MEAS_IN = 0x50,
    REQ_MEAS_CM = 0x51,
    REQ_MEAS_US = 0x52,
    //AUTO_MEAS_CM = 0x54,   //llegir mesura
    REQ_FALSE_IN = 0x56,
    REQ_FALSE_CM = 0x57,
    REQ_FALSE_US = 0x58,
    //AUTO_FALSE_CM = 0x5A, //llegir mesura
    SEND_ULTRA = 0x5C,
    /*
    READ_FIRM = 0x5D,  //llegir mesura
    READ_MEAS = 0x5E,   //llegir mesura
    READ_MIN = 0x5F,   //llegir mesura
    */
    RESET = 0x60,
    SET_ADDR1 = 0xA0,
    SET_ADDR3 = 0xA5,
    SET_ADDR2 = 0xAA

};
#endif
```

8.3.5 SRF02.nc

```
interface SRF02 {

  /**
   * Commands per a la presa de dades
   *
   * @param 0x51 per resultat en cm, 0x52 per resultat en us
   *
   * @return SUCCESS if the set will be performed
   */
  command error_t setCommand( uint8_t val );

  /**
   * Signals the completion of the configuration register set.
   *
   * @param error SUCCESS if the set was successful
   */
  async event void setCommandDone( error_t error );

  command error_t getFirmware();

  async event void getFirmwareDone(error_t error, uint8_t val);


  //command error_t sendPulse();

///////Per fer qualsevol get abans s'ha d'haver fet un setCommand


  /**
   * Return a echo measured in cm.
   *
   * @return SUCCESS if the measurement will be made
   */
  command error_t getEcho();
```

```
  /**
   * Presents the result of the measured echo in cm
   *
   * @param error SUCCESS if the measurement was successful
   * @param val the echo distance
   */
  async event void getEchoDone(error_t error, uint16_t val);


  /**
   * Return a echo measured in cm.
   *
   * @return SUCCESS if the measurement will be made
   */
  command error_t getMinDistance();

  /**
   * Presents the result of the measured echo in cm
   *
   * @param error SUCCESS if the measurement was successful
   * @param val the echo distance
   */
  async event void getMinDistanceDone(error_t error, uint16_t val);

}
```

8.3.6 SRF02C.nc

```
// $Id: Blink.nc,v 1.7 2003/10/07 21:44:45 idgay Exp $

/*                                                      tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement
is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY
OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */
```

```
/**
 * Blink is a basic application that toggles the leds on the mote
 * on every clock interrupt.  The clock interrupt is scheduled to
 * occur every second.  The initialization of the clock can be seen
 * in the Blink initialization function, StdControl.start().<p>
 *
 * @author tinyos-help@millennium.berkeley.edu
 **/
configuration SRF02C {

    provides interface SRF02;

}
implementation {
    components MainC,
               new SRF02P(0xE0) as Sensor,
               new TimerMilliC(),
               new Msp430I2CC(),
               HplMsp430GeneralIOC,
               ActiveMessageC,
               LedsC;

    SRF02 = Sensor;

    MainC.SoftwareInit -> Sensor;
    Sensor.SensorControl -> HplMsp430GeneralIOC.Port35;
    Sensor.Resource -> Msp430I2CC;
    Sensor.Timer -> TimerMilliC;
    Sensor.Leds -> LedsC;
    Sensor.RadioControl -> ActiveMessageC;
    Sensor -> Msp430I2CC.I2CBasicAddr;
    //Msp430Uart0C.Msp430UartConfigure -> Sensor.Msp430UartConfigure;
}
```

8.3.7 SRF02P.nc

```
/*includes SRF02;

#define VECTOR_SIZE 8
#define WAITING_TIME 1000
#define DEFAULT_ADDRESS 0
*/

#include "I2C.h"

generic module SRF02P(uint16_t slaveAddr)
{
    provides {
        interface SRF02;
        interface Init;
    }
    uses {
        interface SplitControl as RadioControl;
        interface I2CPacket<TI2CBasicAddr>;
        interface HplMsp430GeneralIO as SensorControl;
        interface Resource;
        interface Timer<TMilli>;
        interface Leds;
    }
}
implementation {
```

```
    enum {
        STATE_IDLE,
        STATE_SETCOMMAND,
        STATE_READFIRMWARE,
        STATE_READECHO,
        STATE_MINDIST,
        STATE_SENDPULSE
    };

    uint8_t buffer[2];
    uint8_t bufferRead[2];
    uint8_t mState;
    norace uint8_t error_x;
    norace uint16_t value;

    uint8_t size;


/****************** MODULE FUNCTIONS ****************************/

    static error_t doReadReg(uint8_t nextState, uint8_t reg) {
        error_t error = SUCCESS;
        atomic {
            if (mState == STATE_IDLE) {
                mState = nextState;
            }
            else {
                error = EBUSY;
            }
        }
        if (error)
            return error;

        buffer[0] = reg;
        size=1;

        call SensorControl.set();
        //call Timer.startOneShot(500); XXX
        call Resource.request();
        call RadioControl.stop();

        return error;

    }

    static error_t doSetReg(uint8_t nextState, uint8_t reg, uint8_t val) {
        error_t error = SUCCESS;
        atomic {
            if (mState == STATE_IDLE) {
                mState = nextState;
            }
            else {
                error = EBUSY;
            }
        }
        if (error)
            return error;

        buffer[0] = reg;
        buffer[1] = val;
```

```
        size=2;

        call SensorControl.set();
        //call Timer.startOneShot(500); XXX
        call Resource.request();
        call RadioControl.stop();

        return error;
    }

/************** PROVIDED COMMANDS ******************************/


    command error_t Init.init(){

        error_t error = SUCCESS;

        atomic mState = STATE_IDLE;
        call SensorControl.selectIOFunc();
        call SensorControl.makeOutput();
        call SensorControl.clr();
        //call PrintfControl.start();

        return error;
    }

    command error_t SRF02.setCommand(uint8_t val) {
        return doSetReg(STATE_SETCOMMAND, 0, val);
    }

    command error_t SRF02.getFirmware(){
        return doReadReg(STATE_READFIRMWARE, 0);
    }

    command error_t SRF02.getEcho()
    {
        return doReadReg(STATE_READECHO, 2);
    }

    command error_t SRF02.getMinDistance()
    {
        return doReadReg(STATE_MINDIST, 4);
    }



/****************** EVENT IMPLEMENTATION **********************/


    task void returnValues(){

        uint8_t var;

        atomic var= mState;

        switch (var) {
            case STATE_READFIRMWARE:
                signal SRF02.getFirmwareDone(error_x, (uint8_t)value);
                break;
            case STATE_READECHO:
                signal SRF02.getEchoDone(error_x, value);
```

```
                break;
            case STATE_MINDIST:
                //call Leds.led1Toggle();
                signal SRF02.getMinDistanceDone(error_x, value);
                break;
            default:
                break;
        }
        atomic mState = STATE_IDLE;
    }

    event void RadioControl.startDone(error_t error){

        post returnValues();
        return;
    }

    event void RadioControl.stopDone(error_t error){
        return;
    }

    //event void PrintfControl.startDone(error_t error){return;}
    //event void PrintfControl.stopDone(error_t error){return;}


    event void Timer.fired(){

        call Resource.request();
    }

    event void Resource.granted(){

        error_t error = SUCCESS;
        //buffer[0]=0x00;
        error = call I2CPacket.write( ( I2C_START | I2C_STOP ),
                slaveAddr, size, buffer);
        //error = call I2CPacket.write ( ( I2C_START | I2C_STOP ),
slaveAddr,1, buffer);
        return;
    }

    async event void I2CPacket.writeDone(error_t error, uint16_t chipAddr,
            uint8_t len, uint8_t *buf) {

        uint8_t var;

        atomic var=mState;

        switch (var) {
            case STATE_SETCOMMAND:
                atomic mState = STATE_IDLE;
                call Resource.release();
                call SensorControl.clr();
                call RadioControl.start();
                signal SRF02.setCommandDone(error);
                atomic mState = STATE_IDLE;
                break;
            case STATE_READFIRMWARE:
                if(error)
                    signal SRF02.getFirmwareDone(error,0);
                else
```

```
                error = call I2CPacket.read((I2C_START | I2C_STOP),
                        slaveAddr, 1, bufferRead);
            break;
        case STATE_READECHO:
            if (error)
                signal SRF02.getEchoDone(error,0);
            else
                error = call I2CPacket.read((I2C_START | I2C_STOP),
                        slaveAddr, 2, bufferRead);
            break;
        case STATE_MINDIST:
            if (error)
                signal SRF02.getMinDistanceDone(error,0);
            else
                error = call I2CPacket.read((I2C_START | I2C_STOP),
                        slaveAddr, 2, bufferRead);
            break;
        default:
            atomic mState = STATE_IDLE;
            break;
    }
    if (error)
    {
        atomic mState = STATE_IDLE;
        call Resource.release();
        call Leds.led0Toggle();
    }

    //error = call I2CPacket.read(  I2C_START | I2C_STOP ,slaveAddr,10,
buffer2);
    return;
  }



  async event void I2CPacket.readDone(error_t i2c_error, uint16_t chipAddr,
        uint8_t len, uint8_t *buf) {

    /*
    uint8_t i, j;
    uint16_t aux;
    */

    value = buf[0];
    if(len==2)
    {
        value = ( value << 8 & 0xff00);
        value |=  buf[1];
    }
    /**
    if(len==17*2){

        for(i=0, j=0; i<17*2; i=i+2, j++){

            aux = buf[i];
            aux = (aux << 8 & 0xff00);
            aux |= buf[i+1];
            buffer3[j]=aux;
        }
    }
    **/
```

```
        error_x = i2c_error;
        //printf("data: ");
        //for(i=0; i<len;i++) printf(" %u", buf[i]);
        //printf("\n");
        // call PrintfFlush.flush();
        call Resource.release();
        call Leds.led0Toggle();
        call SensorControl.clr();
        call RadioControl.start();
        //call Leds.led1Toggle();

        return;
    }

    //event void  PrintfFlush.flushDone(error_t error){return;}

    default async event void SRF02.setCommandDone(error_t error)
    {
        return;
    }
    default async event void SRF02.getFirmwareDone( error_t error, uint8_t
val)
    {
        return;
    }
    default async event void SRF02.getEchoDone( error_t error, uint16_t val)
    {
        return;
    }
    default async event void SRF02.getMinDistanceDone( error_t error, uint16_t
val)
    {
        return;
    }

}
```